```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 12 16:15:10 2020

@author: jimmy
"""
import numpy as np
import matplotlib.pyplot as plt
import bisect
import time
from astropy import units as u
from astropy import constants as const
from MathTools import EquilTemp

# Class to generate and analze spectral energy distributions (SEDs)
class SED:

    def __init__(self,xvariable,yvariable,r_min=None,r_max=None,dr=1.0,Teff=5780,Rstar=1.0,lambd
        # Inputs:
        #   xvariable: 'freq' or 'wavelen' to determine which Planck form to use
        #   yvariable: 'planck' or 'luminosity' or 'xvar_lumin'
        #   r_min: where disk starts w.r.t star (in au)
        #   r_max: where disk ends w.r.t star (in au)
        #   Teff: effective temperature of host star (in K, default is T_Sun)
        #   Rstar: radius of host star (in Rsun, defualt is 1 R_Sun)
        #   dr: differential radius between rings (in au, default is 1.0)
        #   lambda_min: minimum wavelength to calc. Planck function over (in m)
        #   lambda_max: minimum wavelength to calc. Planck function over (in m)
        #   Number of subintervals to integrate over

        # Cast initial parameters as global variables
        self.xvariable = xvariable
        self.yvariable = yvariable
        self.Teff = Teff*u.K
        self.lambda_min = lambda_min*u.m
        self.lambda_max = lambda_max*u.m
        self.N = N

        # Calculate surface area of sun for later use
        self.sun_SA = 4*np.pi*(Rstar*const.R_sun**2)

        # Define differential radius (distance between rings of disk)
        self.dr = (dr*u.au).to(u.m)

        # If user doesn't specify starting dist. of disk, use r_sub
        if r_min == None:
            # Calculate the dust sublimation radius using rad. equil. temp. eqn.
            self.r_sub = (const.R_sun/2*np.sqrt(1-0.3)*(self.Teff.value/2000)**2).to(u.au)
            self.r_min = self.r_sub.to(u.m)
        # Otherwise, use their starting point and convert it to meters
        else:
            self.r_min = (r_min*u.au).to(u.m)

        # If user doesn't specify ending dist. of disk, use 1000 au
        if r_max == None:
            self.r_max = (1000.0*u.au).to(u.m)
        else:
            self.r_max = (r_max*u.au).to(u.m)

        # Calculate wavelength at which blackbody peaks (in m)
        self.lambda_peak = ((2.90*(10**3)*u.micron*u.K)/self.Teff).to(u.m)

        # Set boundaries of analysis depending on x variable
        if self.xvariable == 'wavelen':

            # Set xdata boundaries and make list of x values (self.x)
```

```
            self.x_min = self.lambda_min
            self.x_max = self.lambda_max
            self.x = np.linspace(self.x_min,self.x_max,self.N)

            # Calculate wavelength at which blackbody peaks (in m)
            self.y_peak = self.lambda_peak

            # Define x-axis label
            self.xlabel = r'$\lambda$ ({0:latex_inline})'.format(self.x_min.unit)

            # Define v-line labels
            self.vlabel = r'Analytical (Planck): $\lambda_{{max}}={0:.2e}$ {1:latex_inline}'.for
            self.num_vlabel = r'Numerical: $\lambda_{{max}}={0:.2e}$ {1:latex_inline}'

            if self.yvariable == 'planck':
                self.ylabel = r'Spectral Radiance per $\Omega$ ({0:latex_inline})'
            elif self.yvariable == 'luminosity':
                self.ylabel = r'$L_{{\lambda}}$ ({0:latex_inline})'
            elif self.yvariable == 'xvar_luminos':
                self.ylabel = r'$\lambda L_{{\lambda}}$ ({0:latex_inline})'

        elif self.xvariable == 'freq':

            # Convert wavelength range to frequency range and make x value list
            self.x_min = self.lambda_min.to(u.s**(-1), equivalencies=u.spectral())
            self.x_max = self.lambda_max.to(u.s**(-1), equivalencies=u.spectral())
            self.x = np.linspace(self.x_max,self.x_min,self.N)

            # Calculate frequency at which blackbody peaks (in s^-1)
            self.y_peak = (5.88*(10**10)*(u.s**(-1))/u.K)*self.Teff

            # Define x-axis label
            self.xlabel = r'$\nu$ ({0:latex_inline})'.format(self.x_min.unit)

            # Define v-line labels
            self.vlabel = r'Analytical (Planck): $\nu_{{max}}={0:.2e}$ {1:latex_inline}'.format(
            self.num_vlabel = r'Numerical: $\nu_{{max}}={0:.2e}$ {1:latex_inline}'

            if self.yvariable == 'planck':
                self.ylabel = r'Spectral Radiance per $\Omega$ ({0:latex_inline})'
            elif self.yvariable == 'luminosity':
                self.ylabel = r'$L_{{\nu}}$ ({0:latex_inline})'
            elif self.yvariable == 'xvar_luminos':
                self.ylabel = r'$\nu L_{{\nu}}$ ({0:latex_inline})'

        else:
            print("Valid entries are 'wavelen' or 'freq'")

    # Function to calculate main part Planck function at given wavelength
    def Planck(self,x,T,units=True):
        # Inputs:
        #   x: value of x-variable to calculate Planck function at
        #   T: temperature of blackbody (in K)
        #   units: boolean to decide if quantities should have units
        #           (no units is preferable if using func. to integrate)
        # Returns:
        #   B: value of Planck function at that wavelength

        # Define temperature with Kelvin units
        #T = T*u.K

        if self.xvariable == 'wavelen':
            # Calculate 2hc^2 (prefactor in Planck's function)
            prefactor = (2*const.h*const.c**2)

            # Calculate hc/kT (constant in exponential of Planck's function)
            exp_factor = (const.h*const.c/(const.k_B*T))
```

```
            if units == False:
                # Calculate value of Planck function at this wavelength
                B = prefactor.value*x.value**(-5)/(np.exp(exp_factor.value/x.value)-1)
            else:
                B = prefactor*x**(-5)/(np.exp(exp_factor/x)-1)

        elif self.xvariable == 'freq':

            # Calculate 2h/c^2 (prefactor in Planck's function)
            prefactor = 2*const.h/const.c**2

            # Calculate h/kT (constant in exponential of Planck's function)
            exp_factor = const.h/(const.k_B*T)

            if units == False:
                # Calculate value of Planck function at this wavelength
                B = prefactor.value*x.value**3/(np.exp(exp_factor.value*x.value)-1)
            else:
                B = prefactor*x**3/(np.exp(exp_factor*x)-1)

        return(B)

    # Function to insert values and sort them
    def insert_list(self,main_list, new_list):
        # Inputs:
        #   main_list: primary list that new_list will be inserted into
        #   new_list: list of new values to insert into main_list
        # Returns:
        #   main_list(updated): primary list with new values correctly sorted

        # Place each value of new_list into correct position
        # main_list.tolist() converts numpy array to reg. list for indexing
        for i in range(len(new_list)):
            bisect.insort(main_list.tolist(),new_list[i])

        return(main_list)

    # Function to plot spectral energy distribution of star
    def SEDStar(self,plot=False):
        # Inputs:
        #   plot: boolean to decide to make plot of SED
        # Returns:
        #   Plot of star's SED
        #   xdata and ydat used to plot SED

        # Determine when function began running
        start_time = time.time()

        # Calculate Planck function at each wavelength/freq
        y = self.Planck(self.x,self.Teff)

        # Calculate total Sun luminosity
        self.star_luminosity = np.pi*self.sun_SA*np.trapz(y,self.x)

        if self.xvariable == 'freq':
            print("Lstar = {0:.3f} Lsun".format(self.star_luminosity/const.L_sun.to(u.J/u.s)))

        # Convert Planck function to luminosity
        if self.yvariable == 'luminosity':
            y *= np.pi*self.sun_SA

        # Convert Planck function to luminosity*xvariable (planck * x**2)
        elif self.yvariable == 'xvar_luminos':
            y *= np.pi*self.sun_SA
            y = np.multiply(self.x,y)
```

```
    elif self.yvariable == 'flux':
        y *= np.pi

    # Find where blackbody peaks from my calculations
    peak_loc = np.argmax(y)
    numerical_max = self.x[peak_loc]

    if self.yvariable == 'planck' and plot == True:
        # Calculate stellar luminosity (np.trapz integrates Planck func.)
        luminosity = np.pi*self.sun_SA*np.trapz(y,self.x)
        print("Luminosity = {0:.3f} Lsun".format(luminosity.to(u.erg/u.s)/const.L_sun.to(u.e

        # Calculate fraction of luminosity from below peak wavelength
        lumin_before = np.pi*self.sun_SA*np.trapz(y[:peak_loc],self.x[:peak_loc])
        frac_before = lumin_before/luminosity*100
        print("{0:.2f}% of energy emitted below peak".format(frac_before))

        # Calculate fraction of luminosity from peak wavelength and beyond
        lumin_after = np.pi*self.sun_SA*np.trapz(y[peak_loc:],self.x[peak_loc:])
        frac_after = lumin_after/luminosity*100
        print("{:.2f}% of energy emitted above peak \n".format(frac_after))

    # Decide whether to plot SED or not
    if plot == True:

        # Create wide figure (w=8in,h=4in)
        plt.figure(figsize=(8,4))

        # Plot data with x-axis on log scale
        plt.plot(self.x,y)
        plt.xscale('log')

        # Plot analytical and numerical wavelength peaks
        plt.vlines(self.y_peak.value,min(y).value,max(y).value,colors='green',\
                   linestyles='dashed',label=self.vlabel)
        plt.vlines(numerical_max.value,min(y).value,max(y).value,colors='red',\
                   linestyles='dashed',label=self.num_vlabel.format(numerical_max.value,nume
        plt.legend()

        # Axes labels and titles
        plt.xlabel(self.xlabel,fontsize=14)
        plt.ylabel(self.ylabel.format(y.unit),fontsize=14)
        plt.title(r'Spectral Energy Distribution for $T_{eff}=$%dK BB'%self.Teff.value,fonts
        plt.tight_layout()

        # Tell user how long function took to run
        end_time = time.time()-start_time
        print('Program took %.2f sec (%.3f min) to run'%(end_time,end_time/60.0))

    return(self.x,y)

# Function to calculated total flux over bandpass
def ResponseFunction(self,bandpass,effic,plot=False):
    # Inputs:
    #   bandpass: frequency ranges of bandpass
    #   effic: efficiencies/response functions of bandpass
    # Returns:
    #   flux_total: total integrated flux over full freq. range

    # Plot efficiency of bandpass
    if plot==True:
        plt.plot(bandpass,effic)

    # Multiply Planck function at each frequency by efficiency
    planck = np.multiply(self.Planck(bandpass,self.Teff),effic)

    # Integrate Planck function over bandpass
```

```python
        flux_total = np.trapz(planck,bandpass)

        return(flux_total)

    def SEDDisk(self,a=.1*10**-3,rho=2.0,plot=False):
        # Inputs:
        #   a: mean grain radius (in m)
        #   rho: mean grain density (in g/cm^3 = kg/m^3)
        # Returns:
        #   disk_ydata: user-selected ydata for disk (planck, luminos, etc.)

        # Define global variables from input variables
        self.grain_size = a*u.m
        self.rho = (rho*u.g/(u.cm**3)).to(u.kg/u.m**3)

        # Calculate mean dust grain mass (kg)
        self.m_grain = 4/3*np.pi*self.grain_size**3*self.rho

        # Calculate grain surface area (m^2)
        self.SA_grain = 4*np.pi*self.grain_size**2

        # Define total disk area (m^2) and mass surface density (kg/m^2)
        self.area_total = np.pi*(self.r_max-self.r_min)**2
        self.surf_dens = const.M_earth/self.area_total

        # Calculate grain surface density (in # of grains/m^2)
        self.grain_dens = self.surf_dens/self.m_grain

        # Make list of radii to describe each ring's distance from star
        radii = np.arange(self.r_min.value,self.r_max.value+self.dr.value,self.dr.value)*u.m

        # Make list of ring areas (2*pi*r*dr)
        areas = 2*np.pi*radii*self.dr

        # Make list of temperatures of each ring
        temps = EquilTemp(radii)

        # Calculate number of grains within each ring
        numGrains = self.grain_dens*areas

        # Calculate mass in each ring and total dust mass in disk
        masses = numGrains*self.m_grain
        dust_mass = np.sum(masses)/const.M_earth # in Earth masses
        print('Total dust mass = {0:.3f} M_Earth'.format(dust_mass))

        # Establish empty array of sums of mono. ydata for each temp
        disk_plancks = []
        disk_luminosities = []

        # Loop over temperatures to calc. Planck at each freq for each temp.
        for i in range(len(self.x)):

            # Track progress of code by printing loop numbers
            #print("Now executing loop {0} of {1}".format(i+1,len(self.x)))

            # Calculate Planck function for each ring
            ring_plancks = numGrains*self.Planck(self.x[i],temps)

            # Calculate monochromatic luminosity for each ring
            ring_luminosities = np.pi*self.SA_grain*ring_plancks

            # Sum Planck + luminosity value of each ring and add sums to lists
            disk_plancks.append(np.sum(ring_plancks))
            disk_luminosities.append(np.sum(ring_luminosities))

        # Recast disk planck and luminosity arrays to better configuration
        disk_plancks = np.asarray([y.value for y in disk_plancks])*disk_plancks[0].unit
```

```
        disk_luminosities = np.asarray([y.value for y in disk_luminosities])*disk_luminosities[0

        # Calculate disk luminosity
        self.disk_luminosity = np.pi*self.SA_grain*np.trapz(disk_plancks,self.x)

        # Save planck array as numpy array w/ proper units
        if self.yvariable == 'planck':
            disk_ydata = disk_plancks

        # Save luminosity array as numpy array w/ proper units
        elif self.yvariable == 'luminosity':
            disk_ydata = disk_luminosities

        # Save nu*L_nu array as numpy array w/ proper units
        elif self.yvariable == 'xvar_luminos':
            disk_ydata = disk_luminosities*self.x

        # Plot SED of disk if requested by user
        if plot == True:
            # Create wide figure (w=8in,h=4in)
            plt.figure(figsize=(8,4))

            plt.plot(self.x,disk_ydata,label=('Disk: ({0:.3f}-{1:.1f}au)'\
                      .format(self.r_min.to(u.au).value,self.r_max.to(u.au).value)))
            plt.xscale('log')

            # Axes labels and titles
            plt.xlabel(self.xlabel,fontsize=14)
            plt.ylabel(self.ylabel.format(disk_ydata.unit),fontsize=14)
            plt.title(r'SED for Disk Around $T_{{eff}}=${0}K Star'.format(self.Teff.value),fonts
            plt.tight_layout()

        return(disk_ydata)

    # Function to plot star and disk SEDs
    def StarDiskProfile(self,plot=True):

        # Determine when function started running
        start_time = time.time()

        # Generate disk ydata values and convert to numpy array
        disk_ydata = self.SEDDisk()#.to(u.erg/u.s)

        # Generate frequencies and star flux values
        xdata,star_ydata = self.SEDStar()
        #star_ydata.to(u.erg/u.s)

        # Calculate sum of disk and Sun flux
        combined_system = np.add(disk_ydata,star_ydata)

        # Calculate ratio of total luminosities of disk and star
        self.luminosity_ratio = self.disk_luminosity.value/self.star_luminosity.value

        if self.xvariable == 'freq':
            print("Disk-to-Star Luminosity Ratio = {0:.2e}".format(self.luminosity_ratio))

        # Plot SEDs if user wants
        if plot == True:
            # Create wide figure (w=8in,h=4in)
            plt.figure(figsize=(8,4))

            # Plotting data
            plt.plot(xdata,disk_ydata,label=r'Disk ({0:.3f}-{1:.1f}au)'\
                     .format(self.r_min.to(u.au).value,self.r_max.to(u.au).value))
            plt.plot(xdata,star_ydata,label='Star')
            plt.plot(xdata,combined_system,label='Both',linestyle='dashed',linewidth=2)
            if self.xvariable == 'freq':
```

```
                plt.plot([],[],'',label=r'$L_{{disk}}/L_{{star}}$ = {0:.2e}'.format(self.luminos
            plt.xscale('log')
            plt.yscale('log')

            if self.xvariable == 'wavelen':
                plt.xlim(xmin=10**-8)
                plt.ylim(ymin=10**-8,ymax=np.max(combined_system).value*10)

            # Axes labels and titles
            plt.xlabel(self.xlabel,fontsize=14)
            plt.ylabel(self.ylabel.format(disk_ydata[0].unit),fontsize=14)
            plt.title(r'SED for $T_{eff}=$%dK Star with Disk'%self.Teff.value,fontsize=18)
            plt.tight_layout()
            plt.legend(prop={'size': 12})

            # Tell user how long function took to run
            end_time = time.time()-start_time
            print('StarDiskProfile took %.2f sec (%.3f min) to run'%(end_time,end_time/60.0))

        return(combined_system.value)
"""
# Code to test class and functions
test = SED('freq','xvar_luminos',r_min=1.0,N=10**4)
#test.Planck(10**-6*u.m,units=False)
#test.SEDStar(True)
test.StarDiskProfile(plot=True)
#"""
```