2. Continuing Introduction to Python

J. S. Wright
jswright@tsinghua.edu.cn

2.1 Discussion of Problem Set 1

Your task in Problem Set 1 was to create a function that took an array of wavelengths and a single temperature as inputs, and provided as output the intensity at those wavelengths due to emission by a blackbody of that temperature. The relevant equation is the Planck function:

$$B(\lambda, T) = \frac{2hc^2}{\lambda^5} \frac{1}{\exp\left(\frac{hc}{\lambda k_B T}\right) - 1}$$

where $h = 6.625 \times 10^{-34}$ J s is the Planck constant, $c = 3 \times 10^8$ m s^{-1} is the speed of light, and $k_B = 1.38 \times 10^{-23}$ J K^{-1} is the Boltzmann constant. One possible solution is the following:

```
1  import numpy as np
2
3  def PlanckFunc(wl,T):
4      '''
5      Evaluate the emission intensity for a blackbody of temperature T
6      as a function of wavelength
7
8      Inputs:
9          wl :: numpy array containing wavelengths [m]
10         T  :: temperature [K]
11
12     Outputs:
13         B  :: intensity [W sr**-1 m**-2]
14     '''
15     wl = np.array(wl) # if the input is a list or a tuple, make it
           an array
16     h  = 6.625e-34    # Planck constant [J s]
```

```
    c  = 3e8          # speed of light [m s**-1]
    kb = 1.38e-23     # Boltzmann constant [J K**-1]
    B  = ((2*h*c*c)/(wl**5))/(np.exp((h*c)/(wl*kb*T))-1)
    return B
```

Some of you may have encountered a message like:

```
In [1]: %run "planck.py"
planck.py:19: RuntimeWarning: overflow encountered in exp
B  = ((2*h*c*c)/(wl**5)) * 1./(np.exp((h*c)/(wl*kb*T))-1)}
```

What this means is that the function `numpy.exp()` cannot return a valid result because one or more inputs is either too large or too small (in this case it is most likely too large). A warning like this may or may not indicate an important problem. For example, you may have received this message if you tried wavelength inputs less than about 10 nm or so. Emission at these wavelengths is vanishingly small for blackbodies at these temperatures. In this case, we can ignore the warning because it isn't relevant to the results that we are interested in. You may also have received the message if you made an error in inputting the value for one of the constants (e.g., $h = 6.625 \times 10^{-5}$ instead of 6.625×10^{-34}). In this case we need to pay attention to the warning because it indicates a fundamental problem with our code. Identifying the source of and reason behind warnings is an important step toward being confident that our code is correct.

One of the more difficult aspects of this assignment was how to specify the array of wavelengths. For consistency with the parameters, the wavelengths must be provided in SI units (i.e., meters). One option is to use the intrinsic `range` function to generate a list of integers, and then to modify that list of integers after converting it to a numpy array:

```
In [2]: wavelengths = np.array(range(100000))*1e-9
```

This solution works, but numpy offers several more convenient alternatives. For example, `np.arange()` combines the first two functions, creating a numpy array that contains the specified range:

```
In [3]: wavelengths = np.arange(100000)*1e-9
```

Using `numpy.arange()` also eliminates the requirement that `range()` can only return integer lists. We can get the same result by writing:

```
In [4]: wavelengths = np.arange(0,1e-4,1e-9)
```

All of the above options include zero (which is not a useful feature in this case), and none of them include 1×10^{-4} (i.e., 100 μm). We could correct for this by adding `1e-9` to any of the above examples, or we could get the equivalent result by using numpy's `linspace` function:

```
In [5]: wavelengths = np.linspace(1e-9,1e-4,100000)
```

The inputs in this example specify the first element in the array, the last element in the array and the total number of elements in the array. For example, if we think that 100000 elements is too many and we want to reduce the number of elements by a factor of 100, we could cover the same range by writing:

```
In [6]: wavelengths = np.linspace(1e-9,1e-4,1000)
```

The best option for ranges that cover several orders of magnitude, however, is to use the related numpy function `logspace`:

```
In [7]: wavelengths = np.logspace(-9,-4,1000)
```

This command creates an array that spans the range from `1e-9` (1 nm, in our case) to `1e-4` (100 μm), with elements at equal intervals in $\log_{10}$ space (rather than at equal intervals in linear space). It is also possible to specify a base other than 10; for instance, you could generate an array containing the first nine integer powers of 2 by typing:

```
In [8]: np.logspace(0,8,9,base=2)
Out[8]: array([ 1.,  2.,  4.,  8., 16., 32., 64., 128., 256.])
```

Some of you found other useful shortcuts. For example, the `scipy` module contains a number of commonly used constants (including h, c and k_B), which can be accessed by importing `scipy.constants`:

```
import numpy as np
import scipy.constants as spc

def PlanckFunc(wl,T):
    '''
    Evaluate the emission intensity for a blackbody of temperature T
    as a function of wavelength

    Inputs:
        wl :: numpy array containing wavelengths [m]
        T  :: temperature [K]

    Outputs:
        B  :: intensity [W sr**-1 m**-2]
    '''
    wl = np.array(wl) # if the input is a list or a tuple, make it
        an array
    B  = ((2*spc.h*spc.c**2)/(wl**5))/(np.exp((spc.h*spc.c)/(wl*spc.
        k*T))-1)
    return B
```

You will learn in this course (if you haven't already) that there are many ways to write a program, but no one "right" way. We should aspire to write fast programs that are easy to read and give the most accurate possible results, but these objectives may sometimes conflict. For example, using the constants from `scipy.constant` improves the accuracy of the parameters, with clear impacts on the results:

```
In [9]: from planck import PlanckFunc as PF1
In [10]: from planck_spc import PlanckFunc as PF2
In [11]: PF1(10e-6,255)
Out[11]: 4218277.9779265849
In [12]: PF2(10e-6,255)
Out[12]: 4237074.7368364623
```

On the other hand, using the constants from `scipy.constant` makes the program more difficult to read and understand if we have not seen it before. In the initial code we had comments that clearly defined the constants and specified the associated units; now we would have to access the documentation for `scipy.constant` using `help(spc)` to find the same information. This is not a big deal in this particular case, but it is easy to see how we might have to make choices when writing more complicated programs. As a programmer, it is your job to make informed choices: given what I know about who will use this program, what approach will be most useful?

The extension of this function to lists is relatively straightforward using the framework outlined in notes 1 (see, e.g., `planck_list.py` among the scripts for this week). I want to also highlight an alternative solution, in which we use list comprehension rather than the more familiar loop block structure:

```
from math import exp

def PlanckFunc(wl,T):
    '''
    Evaluate the emission intensity for a blackbody of temperature T
    as a function of wavelength

    Inputs:
        wl :: list containing wavelengths [m]
        T  :: temperature [K]

    Outputs:
        B  :: intensity [W m**-2]
    '''
    h  = 6.625e-34     # Planck constant [J s]
    c  = 3e8           # speed of light [m s**-1]
    kb = 1.38e-23      # Boltzmann constant [J K**-1]
    B  = [((2*h*c**2)/(l**5))/(exp(h*c/(T*kb*l))-1) for l in wl]
    return B
```

This code does exactly the same thing as if we wrote a loop and nested the calculation within it, appending or writing to the output list at every step. List comprehension can be quite useful for writing concise code that does not depend on numpy.

We can use any of these functions to calculate the blackbody emission as a function of wavelength for $T = 6000$ K and $T = 255$ K. Assume we have saved our function in a file called `planck.py`, which includes the function `PlanckFunc` (and potentially other items). We can then write a short script to import and apply that function:

```
import numpy as np
from planck import PlanckFunc

# one of the more difficult problems is how to specify the wavelengths
#  (we want micrometers between about 0.1 and 100, but function needs meters)
#
# Option 1: use range to create a list, and convert to an array
lamdas = np.array(range(100000))*1e-9
# Option 2: list comprehension
lamdas = [i*1e-9 for i in range(100000)]
# Option 3: use the numpy.arange function to create an array range directly
lamdas = np.arange(100000)*1e-9
lamdas = np.arange(0,1e-4,1e-9)
# Option 4: use the numpy.linspace function to create a more convenient (and smaller) array
lamdas = np.linspace(1e-9,1e-4,1000)
# Option 5: use numpy.logspace to create an evenly spaced range in log base 10
lamdas = np.logspace(-8,-2,1000)

B_6000 = PlanckFunc(lamdas,6000)
B_255  = PlanckFunc(lamdas,255)
```

This script shows that not only can we import functions and other data from modules included in our python installation, we can also import functions and other data from our own scripts (see also lines `In [9]` and `In [10]` above). We can only do this with scripts that are in python's current search path. The easiest way to do this is to put these scripts into the current working directory. We could also create a folder to store our most useful scripts and add this to the global python search path, which is typically in the environment variable `PYTHONPATH`. In most cases, it is safer to import the `sys` module and temporarily add directories to `sys.path`, a list that includes all of the directories to search:

```
In [13]: import sys
In [14]: sys.path.append('/path/to/my/scripts')
```

Python searches the directories in `sys.path` in order. As a result, if your script has the same name as another module, then you could instead use `insert` to put the directory containing your script at the front of `sys.path`:

```
In [15]: sys.path.insert(1,'/path/to/my/scripts')
```

This can sometimes cause unexpected behavior, which is why it is better to modify `sys.path` (which only affects the current session) than to modify the global environment variable `PYTHONPATH` (which affects this and every future session). See this page and this page for more details. The advantages of not having to copy and paste functions that you want to

include in multiple scripts are obvious. When you import a module that you have created yourself, python will create a compiled version of your code in the file `mymodule.pyc`. This speeds up subsequent access to the functions in the module.

You may have noticed in writing your code that `lambda` is a protected word in python (i.e., we should not name our vector of wavelengths `lambda`). The `lambda` construct is a tool for **functional programming** that allows us to create small anonymous functions without constructing specific function definitions (often in combination with list comprehension). We may deal with this construct later in the course – I don't personally use it often, but many people find it quite useful. If you are interested, you can learn more about it at this page.

The other potentially challenging part of Problem Set 1 was plotting the two intensity distributions on the same axes, because their magnitudes are substantially different. One solution is to normalize each distribution by its maximum, as shown in Fig. 2.1:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8.3,5),dpi=300)
fig.subplots_adjust(bottom=0.1,top=0.9,left=0.1,right=0.95)
ax1  = fig.add_subplot(1,1,1)
ax1.fill_between(lamdas,np.zeros(lamdas.shape),
            y2=B_6000/B_6000.max(),color='#e6ab02',alpha=0.2)
pl1l = ax1.plot(lamdas,B_6000/B_6000.max(),color='#e6ab02',
            linewidth=2,label='6000 K')
ax1.fill_between(lamdas,np.zeros(lamdas.shape),
            y2=B_255/B_255.max(),color='#377eb8',alpha=0.2)
pl2l = ax1.plot(lamdas,B_255/B_255.max(),color='#377eb8',
            linewidth=2,label='255 K')
ax1.set_xlim((1e-7,1e-4))
ax1.set_xscale('log')
ax1.set_xticks([1e-7,1e-6,1e-5,1e-4])
ax1.set_xticklabels(['0.1','1','10','100'])
ax1.set_xlabel(u'Wavelength [\u00b5m]')
ax1.set_ylim((0,1.05))
ax1.set_yticks(np.arange(0,1.1,0.2))
ax1.set_ylabel(r'Normalized Intensity')
lgd = ax1.legend(fancybox=True,loc=1)
plt.savefig(fdir+'planck1.pdf')
```

We can access the maximum value of a numpy array by `arrayname.max()`. Sometimes an array may have more than one dimension, in which case we can also use the `axis` keyword:

```
In [16]: a = np.array([[1,2],[3,4]])
In [17]: a
Out[17]:
array([[1, 2],
       [3, 4]])
In [18]: a.max(axis=0)
Out[18]: array([3, 4])
```

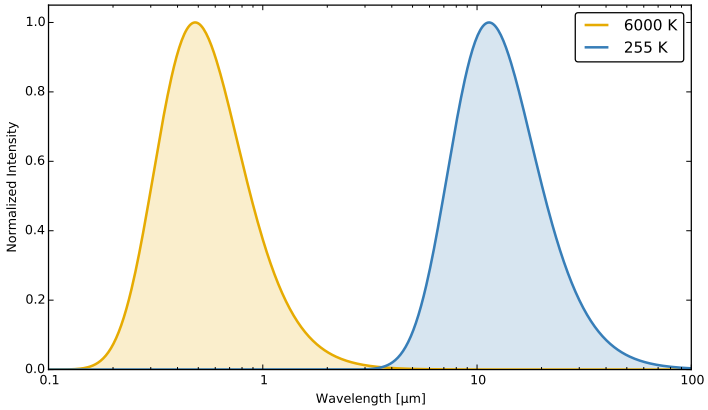

Figure 2.1: We can normalize each distribution by its maximum intensity to fit both plots on the same axis.

```
In [19]: a.max(axis=1)
Out[19]: array([2, 4])

In [20]: a.max()
Out[20]: 4
```

We can also find the minimum value using `a.min()`, or access the index of the maximum or minimum value (`a.argmax()` and `a.argmin()`, respectively). Note that these last two functions provide the index as though the array were flat (i.e., one-dimensional): for instance, `a.argmax()` returns 3 rather than `(1,1)`. Both `a.argmax()` and `a.argmin()` also take the axis keyword.

Two other elements of Fig. 2.1 are worth noting. First, I have given the x-axis a logarithmic scale rather than a linear one. We can switch back and forth using `axs.set_xscale('log')` and `axs.set_xscale('linear')`. Matplotlib also includes the convenience function `semilogx()`, which many of you used — this function is similar to `plot()`, but it automatically applies a logarithmic scale to the x-axis. Second, I have used `axs.fill_between()` to fill the area between each line and the y-axis (here represented by `np.zeros(lamdas.shape)`, which returns an array of the same shape as `lamdas` that is everywhere equal to 0). This code would work just as well if we simply use `0` rather then `np.zeros(lamdas.shape)`; I use `np.zeros(lamdas.shape)` for two reasons: to clarify what the code is doing and to illustrate the use of `np.zeros()`. The syntax for this plot object is fairly straightforward, with the possible exception of the keyword `alpha=0.2`. This keyword is common among many `pyplot` objects, and is used to set the transparency. For those familiar with image editing software such as Adobe Photoshop, setting `alpha=0.2` for an object means that the object will have an opacity of 20% (i.e., it will be mostly transparent). Setting `alpha=1` will cause the object to appear as normal (completely opaque), while setting `alpha=0` will cause the object

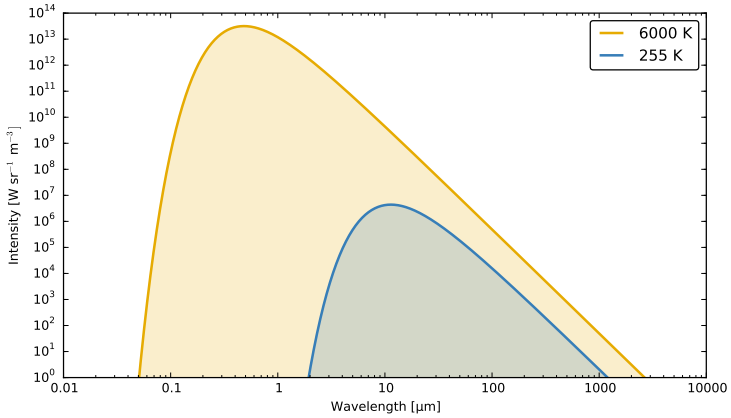

Figure 2.2: Using `axs.set_yscale('log')` allows us to fit both plots on the same axes, while also giving a slightly different perspective on the relative intensity at different wavelengths.

to become completely transparent (in which case it won't appear at all). The `alpha` keyword does not always have an impact on the final graphics. For example, files saved in encapsulated postscript (.eps) format do not retain transparency information.

There are other solutions to the difference in scale between the intensity of a blackbody at 6000 K and the intensity of a blackbody at 255 K. For example, just as we applied a logarithmic scale to the x-axis using `axs.set_xscale('log')`, we can apply a logarithmic scale to the y-axis using `axs.set_yscale('log')`:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8.3,5),dpi=300)
fig.subplots_adjust(bottom=0.1,top=0.9,left=0.1,right=0.95)
ax1  = fig.add_subplot(1,1,1)
ax1.fill_between(lamdas,np.zeros(lamdas.shape),y2=B_6000,
            color='#e6ab02',alpha=0.2)
ax1.plot(lamdas,B_6000,color='#e6ab02',linewidth=2,label='6000 K')
ax1.fill_between(lamdas,np.zeros(lamdas.shape),y2=B_255,
            color='#377eb8',alpha=0.2)
ax1.plot(lamdas,B_255,color='#377eb8',linewidth=2,label='255 K')
ax1.set_xlim((1e-8,1e-2))
ax1.set_xscale('log')
ax1.set_xticks([1e-8,1e-7,1e-6,1e-5,1e-4,1e-3,1e-2])
ax1.set_xticklabels(['0.01','0.1','1','10','100','1000','10000'])
ax1.set_xlabel(u'Wavelength [\u00b5m]')
ax1.set_ylim((1,1e14))
ax1.set_yscale('log')
ax1.set_ylabel(r'Intensity [W sr$^{-1}$ m$^{-3}]$')
lgd = ax1.legend(fancybox=True,loc=1)
plt.savefig(fdir+'planck2.pdf')
```

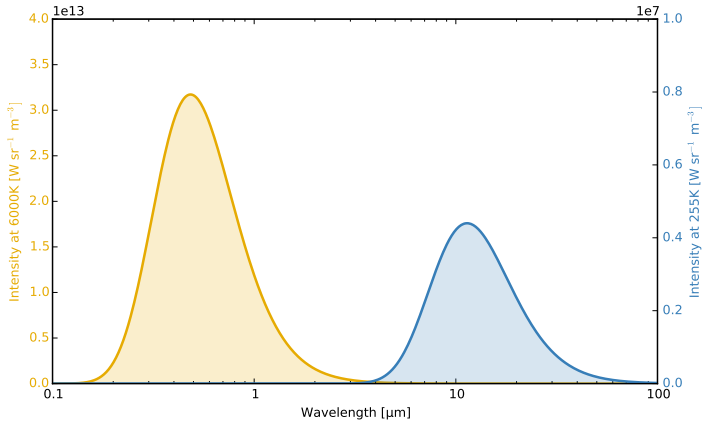

Figure 2.3: We can use `axs.twinx()` to fit both plots on the same x-axis without changing the range or scale of the y-axis.

Just as with `np.logspace()`, we use this approach to allow the y-axis to cover many orders of magnitude (in this case, 14). This approach also has the added benefit of providing a slightly different perspective on the results: Figure 2.2 emphasizes that the intensity of emission from a 6000 K blackbody is still greater than the intensity of emission from a 255 K blackbody at long wavelengths, even though the maximum intensity is shifted to much shorter wavelengths for the warmer blackbody. Note that like `semilogx()`, matplotlib provides the convenience function `loglog()`, which automatically applies a logarithmic scale to both axes.

Matplotlib has excellent text support, including access to LaTeX and unicode formatting. Both types of formatting are included in the previous example. The command `ax1.set_ylabel()` in line 19 uses LaTeX formatting, as indicated by the leading `r` before the body of the string, which enables the superscripts and replaces the hyphen - with the mathematical − sign. Code enclosed by dollar signs (`$...$`) is processed as LaTeX markup. Note that the font used for LaTeX markup is different from the font used for other text. If this bothers you, you can use `\mathregular{}` to enclose text that you want to show in the standard font. Line 16 provides an example of a unicode string, which is preceded by a leading `u`. Here I used unicode tables to look up the unicode representation of the character μ (00b5) so that we can include the units μm in the x-axis label. For more extensive changes (e.g., to completely change the font, or to enable support for Chinese characters) we can use the matplotlib font manager:

```
1 # -*- coding: utf-8 -*-
2
3 from matplotlib.font_manager import FontProperties
4 KaiTi = FontProperties(fname='/Library/Fonts/Kaiti.ttc')
5
6 ax1.set_title(u'[Chinese text here]',fontproperties=KaiTi)
```

For Chinese character support, both the first line (`# -*- coding: utf-8 -*-`) and the

leading `u` in the string are necessary to specify that extensive access to the unicode tables is needed. Less invasive options include changing the characteristics of the font using keywords like `family`, `style`, and `weight` (see this page for details), or using the convenience functions supplied by the `seaborn` module. We will discuss many other aspects of `seaborn` in future lectures.

Many of you used a third approach; namely, creating a second y-axis on the right of the plot using `axs.twinx()`:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8.3,5),dpi=300)
fig.subplots_adjust(bottom=0.1,top=0.9,left=0.1,right=0.9)
ax1  = fig.add_subplot(1,1,1)
ax1.fill_between(lamdas,0,y2=B_6000,color='#e6ab02',alpha=0.2)
pl1l = ax1.plot(lamdas,B_6000,color='#e6ab02',linewidth=2)
ax1.set_xlim((1e-7,1e-4))
ax1.set_xscale('log')
ax1.set_xticks([1e-7,1e-6,1e-5,1e-4])
ax1.set_xticklabels(['0.1','1','10','100'])
ax1.set_xlabel(u'Wavelength [\u00b5m]')
ax1.set_ylim((0,4e13))
ax1.set_ylabel(r'Intensity at 6000K [W sr$^{-1}$ m$^{-3}]$',
            color='#e6ab02')
for tl in ax1.get_yticklabels(): tl.set_color('#e6ab02')
ax2 = ax1.twinx()
ax2.fill_between(lamdas,0,y2=B_255,color='#377eb8',alpha=0.2)
pl2l = ax2.plot(lamdas,B_255,color='#377eb8',linewidth=2)
ax2.set_xlim((1e-7,1e-4))
ax2.set_xscale('log')
ax2.set_xticks([1e-7,1e-6,1e-5,1e-4])
ax2.set_xticklabels(['0.1','1','10','100'])
ax2.set_xlabel(u'Wavelength [\u00b5m]')
ax2.set_ylim((0,1e7))
ax2.set_ylabel(r'Intensity at 255K [W sr$^{-1}$ m$^{-3}]$',
            color='#377eb8')
for tl in ax2.get_yticklabels(): tl.set_color('#377eb8')
plt.savefig(fdir+'planck3.pdf')
```

This approach also works well, but make sure that the range of your x-axis fits both plots! Some of you cut off the intensity distribution for the 255 K blackbody in the middle. It can also be helpful to visually emphasize the difference in scale between the two distributions (see Fig. 2.3) — if you're not careful about this, others who see your plot might mistakenly think that the peak intensity is larger for the 255 K black body than for the 6000 K black body!

One of you even tried (unsuccessfully) to create an animation showing how the emission changes as a function of temperature. This is an ambitious attempt for the first homework assignment, but as we are unlikely to use this feature later in the course, a working example is shown below:

```
import matplotlib.pyplot as plt
```

```
import matplotlib.animation as manimation

# data arrays (wavelengths for all plots, emission temperatures for
    each plot)
wl    = np.logspace(-8,-2,1000)
T     = np.linspace(255,6000,100)

# instantiate writer
FFMpegWriter = manimation.writers['ffmpeg']
writer = FFMpegWriter(fps=15)

# set up plot
fig = plt.figure()
plt.plot(wl,PlanckFunc(wl,T[-1]),color='#e6ab02',linewidth=2)
plt.plot(wl,PlanckFunc(wl,T[0]),color='#377eb8',linewidth=2)
l, = plt.plot([], [], '-', color='#666666',linewidth=2)
plt.xlim(1e-8,1e-2)
plt.ylim(0,1e14)
plt.xscale('log')
plt.xticks([1e-8,1e-7,1e-6,1e-5,1e-4,1e-3,1e-2],['0.01','0.1','1','
    10','100','1000','10000'])
plt.xlabel(u'Wavelength [\u00b5m]')
plt.ylim((1,1e14))
plt.yscale('log')
plt.ylabel(r'Intensity [W sr$^{-\mathregular{1}}$ m$^{-\mathregular
    {3}}]$')

# write movie
outf = '/Users/jswright/courses/AtmosphereOceanInteractions/stable/
    assignments/01ebm/animation1.mp4'
with writer.saving(fig, outf, len(T)):
    for i in range(len(T)):
        B = PlanckFunc(wl,T[i])
        l.set_data(wl, B)
        writer.grab_frame()
```

The output is included in the archive of scripts for this lecture. This script requires installing the ffmpeg package (for example, by downloading pre-built binaries appropriate for your system and putting them in your python executable directory; e.g., `$homedir/anaconda/bin`). The procedure for doing this varies by system, but you can find the ffmpeg source code and various binaries here and here.

2.2 Basic Dynamical Modeling in Python

Python can be used to build numerical models of dynamical systems. We will use the famous model introduced by Lorenz (1963) to illustrate this process. Lorenz proposed the following system of three equations as a simple model of convection in a fluid heated from below:

$$\frac{dX}{dt} = \sigma(Y - X)$$
$$\frac{dY}{dt} = -XZ + rX - Y$$
$$\frac{dZ}{dt} = XY - bZ$$

The variable X represents the speed of the fluid, the variable Y represents the temperature difference between rising and sinking portions of the fluid, and the variable Z represents the vertical temperature gradient. The variables X, Y and Z are dependent variables, with the time t the independent variable. All of these variables are dimensionless by definition. The equations also include three parameters: σ is the Prandtl number, r is the Rayleigh number, and b is the width to height ratio of the fluid container. This dynamical system is a chaotic attractor (meaning that small differences in initial conditions are amplified in the solution). The extent of this amplification (i.e., the predictability) depends on the current state of the system: differences grow rapidly in some parts of the system, but slowly in others. The Lorenz system is a useful illustration of the complications inherent in weather forecasting (where predictions and predictability are also heavily dependent on current conditions), as well as the difference between climate (the shape of the attractor) and weather (the evolving location of the trajectory in time).

The Lorenz system can be modeled in python using a function like:

```
def lfunc(t,y,b,r,s):
    '''
    Rates of change for x,y,z in Lorenz 1963 model

    Inputs:
        t :: independent variable
        y :: dependent variables
        b :: parameter representing geometric factor
        r :: parameter representing Rayleigh number
        s :: parameter representing Prandtl number
    '''
    xdot = -s*y[0] + s*y[1]
    ydot = -y[0]*y[2] + r*y[0] - y[1]
    zdot = y[0]*y[1] - b*y[2]
    return [xdot,ydot,zdot]
```

The structure of this function is meaningful, as we will see later. The function takes as inputs the independent variable `t`, a data structure `y = [X, Y, Z]` containing the dependent variables, and three parameters. We can integrate this system of ordinary differential equations numerically to approximate a solution for a given set of initial conditions. The most intuitive way is to apply a forward Euler method, as in the following example:

```
import numpy as np
from lorenz import lfunc

def lorenz_euler(xyz0=(0.,1.,1.05),dt=0.01,nsteps=10000,b=(8.0/3.0),
    r=28.0,sigma=10.0):
    '''
    Integrate Lorenz 1963 model using forward (explicit) Euler

    Optional keywords:
        xyz0   :: tuple containing initial conditions for x, y, z
        dt     :: time step (should be 0.01 or less)
        nsteps :: number of time steps
        b      :: geometric factor
        r      :: Rayleigh number
        sigma  :: Prandtl number

    Output:
        3 x nsteps array containing the solution
    '''
#-- initialize solution
    xyz = np.empty((3,nsteps+1))
    xyz[:,0] = xyz0
#-- step through time
    for ii in range(nsteps):
        xdot,ydot,zdot = lfunc(ii,xyz[:,ii],b=b,r=r,sigma=sigma)
        xyz[0,ii+1] = xyz[0,ii] + xdot*dt
        xyz[1,ii+1] = xyz[1,ii] + ydot*dt
        xyz[2,ii+1] = xyz[2,ii] + zdot*dt
#-- return solution
    return xyz

xyz = lorenz_euler()
```

The magnitude of the time step `dt` is critically important. If this magnitude is too large, the numerical solution will be unstable and the results will fail to converge to the solution. If it is too small, then the code will run very slowly. For the Lorenz model, a time step of approximately 0.01 is ideal. A description of the mathematics behind this is beyond the scope of this course, but if you write a numerical model and find that the solution blows up toward infinity (or negative infinity), you may want to try decreasing your time step. One approach that can be useful is to use sub-time steps. For example, if convergence requires a time step of 0.01 s but you only need the output of the model on a time step of 1 s, you can discretize each 1,s time step into 100 sub-time steps. Note that this may require linear approximations for parameters that are only available on the coarser time grid.

Once we have a numerical solution, we can plot it using `matplotlib.pyplot`. The Lorenz system involves three variables,so we can try plotting on a three dimensional axis:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

```

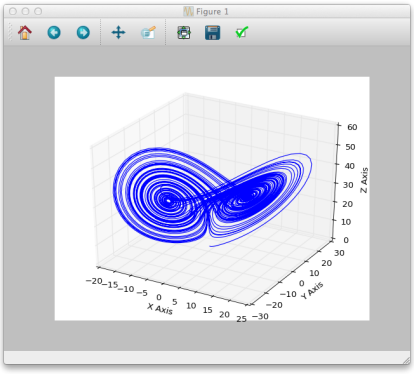

Figure 2.4: A forward Euler solution to the Lorenz equations.

```
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(xyz[0,:], xyz[1,:], xyz[2,:])
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
plt.show()
```

This script produces a plot of the famous Lorenz butterfly (Fig. 2.4). More information about three-dimensional plotting capabilities in matplotlib can be found in the mplot3d tutorial. We will explore alternative methods visualizing three-dimensional data (such as contour plots, color meshes and color norms) later in the course.

Those of you who are familiar with Matlab may be accustomed to using `ode45` or other built-in **integrators** to generate numerical solutions for systems of ordinary differential equations. Python integrators are provided by the `scipy.integrate.ode` interface. The exact analogue for `ode45` is accessed using the 'dopri5' integrator, as shown in the following example:

```
import numpy as np
from lorenz import lfunc
from scipy.integrate import ode

def lorenz_scipy(xyz0=(0.,1.,1.05),dt=0.01,nsteps=10000,b=(8.0/3.0),
                 r=28.0,sigma=10.0,method='dopri5'):
    '''
    Integrate Lorenz 1963 model using scipy.integrate.ode

    Optional keywords:
        xyz0   :: tuple containing initial conditions for x, y, z
        dt     :: time step
        nsteps :: number of time steps
        b      :: geometric factor
        r      :: Rayleigh number
        sigma  :: Prandtl number
```

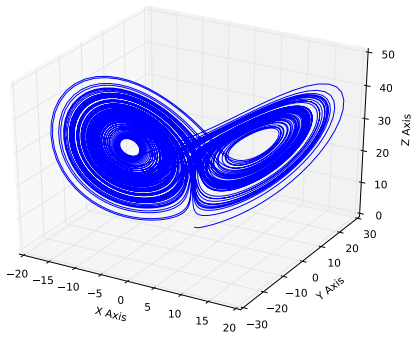

Figure 2.5: A solution to the Lorenz equations using `scipy.integrate.ode`.

```
17          method :: integrator to use
18
19      Output:
20          3 x nsteps array containing the solution
21      '''
22      xyz = np.empty((3,nsteps))
23      xyz[:,0] = xyz0
24      lf = ode(lfunc).set_integrator(method)
25      lf.set_initial_value(xyz0,0).set_f_params(b,r,sigma)
26      t1 = dt*nsteps
27      ii = 0
28      while lf.successful() and lf.t <= t1:
29          lf.integrate(lf.t+dt)
30          xyz[:,ii] = lf.y
31          ii += 1
32      return xyz
33
34  xyz = lorenz_scipy()[0]
```

The syntax for the `ode` interface is descriptive. For example, lines 24 and 25 of the above example could also be written as

```
In [21]: lf = ode(lfunc)
In [22]: lf.set_integrator('dopri5')
In [23]: lf.set_initial_value(xyz0,0)
In [24]: lf.set_params(b,r,sigma)
```

The first line instantiates `ode` and tells it to integrate the system of equations described in the function `lfunc`. The second line tells `ode` to use the `'dopri5'` integration method, an explicit Runge–Kutta method of order (4)5. The third line tells `ode` to use the data in `xyz0` as the initial values for the three dependent variables and `0` as the initial value for the independent variable `t`. The fourth line tells `ode` what values to use for the parameters `b`, `r` and `s` in the

function `lfunc`. Further details and alternative integration methods are provided in the scipy documentation for `ode`.

The results of the integration using `scipy.integrate.ode` are shown in Fig. 2.5. Even though the initial conditions and the time step are identical, the solution using `ode` is not the same as the solution using the forward Euler solution. These differences show the potential effects of differences in accuracy and rounding error between the two approaches. The numerical integration methods provided by `scipy.integrate.ode` are precompiled, and will therefore be faster than equivalent methods using only python code.

2.3 Basic Data Input: The csv Module

Python allows for data input from and output to a wide variety of file types. Perhaps the simplest of these file types is **'comma separated values'**, or CSV, which are accessible via the `csv` module. Although we won't cover data output at this time, I will show and briefly discuss a few examples of data input using `csv`. Please refer to the csv documentation for further details.

The first example reads in a simple CSV file with four columns and a header row. The header row indicates the variables contained in each column; namely, a file identifier, the time (in seconds since midnight universal time on 1 January 1970), the modified Julian date, and MODIS observations of enhanced vegetation index (EVI) averaged over the southern Amazon. Each column is separated by a comma. To read in this data, we could use a script like:

```
1  import csv
2
3  # data directory
4  mdir = '/Users/jswright/projects/IsotopesAmazon/modis/'
5  # variables to hold the data
6  date = []
7  evi  = []
8  # open the CSV file
9  with open(mdir+'mcd43_pooled_all.csv') as csvfile:
10     reader = csv.reader(csvfile, delimiter=',')
11 #-- skip the header line
12     reader.next()
13     for row in reader:
14 #------ save the date (a string) from the third column
15         date.append(int(row[2]))
16 #------ save the EVI (a float) from the fourth column
17         evi.append(float(row[3]))
```

The script starts by importing the `csv` module and specifying the location of the data. We then create empty lists to hold the Julian date for each measurement and the associated value of EVI.

Data input begins in line 9. Here, we define the beginning of a block of code using the syntax `with open(filename) as csvfile:`. The file is open for the entirety of this block of code, with access to the file terminated at the end of the block of code (which is also the end of the indented section). The following line instantiates a `csv.reader` object from `csvfile`, with

the **delimiter** `','`. The delimiter should correspond to whatever symbol marks the separation between columns. Frequently used delimiters include commas (`','`), semicolons (`';'`), tab characters (`'\t'`) and empty spaces (`' '`).

After instantiating the `reader` object, we can cycle through it. We are not interested in the header line, so we skip it in line 12 using the intrinsic function `reader.next()`. We then loop through reader by row, writing the Julian date from the third column into the list `date` and the EVI from the fourth column into the list `evi`. Note that we cannot go backwards in a `csv.reader` object, nor can we index it directly. We can only loop through from the beginning of the file to end.

All of the elements of rows in a `csv.reader` object are strings, and so they must be converted to integers or floats if we want the data in numeric form. If we specify the wrong delimiter (for instance, if we specify `'\t'` when the delimiter is actually `' '`), then row will be one long string containing all of the columns at once and we cannot convert it to a numeric form directly. Sometimes when the delimiter is `' '`, `row` will contain a large number of empty strings (because there may be several spaces between columns rather than only one). The following example shows how to deal with this problem:

```
1  import csv
2  import numpy as np
3
4  # data directory and input file
5  ddir = '/Users/jswright/projects/Aerosols/ConvectiveTransport/data/'
6  tfil = ddir+'congo/convective_minus_background_aerosol.txt'
7
8  prfl = []
9  with open(tfil) as csvfile:
10     reader = csv.reader(csvfile, delimiter=' ')
11     for row in reader:
12 #------ filter empty spaces and convert to floats
13         row = np.array(filter(None,row)).astype('float')
14 #------ append tuple containing case identifier and profile
15         prfl.append((int(row[19]),row[:16]))
16
17 # use zip() to retrieve a list of the case IDs...
18 case = list(zip(*prfl)[0])
19 # ...or an n by z array containing all n profiles
20 prof = np.array(zip(*prfl)[1])
```

In line 13, we use the intrinsic function `filter()` to remove all instances of `None` from the list `row`, leaving only the columns. This procedure removes empty strings because empty strings evaluate to `None`. We also convert row to a numpy array and convert all of the elements to floating point numbers. This approach is sometimes more convenient than converting each column individually, provided you want to convert all of the columns (you can also use a slice if there are columns that you don't wish to convert). This script also introduces the intrinsic function `zip()`, which is convenient for separating lists of tuples (or lists of lists) into their constituent elements. In this case we have created a list of tuples. Each tuple contains a case identifier (in index 0) and a vertical profile (in index 1). Using `zip()` we can easily create a list that contains only the identifiers (line 18) or an array that contains all of the profiles (line 19).

The use of the `csv` module and its `reader` object requires knowledge of the underlying file. CSV files can include large numbers of header lines, which can make reading them somewhat complex. The following example needs to access metadata from only a few header lines, and the code is already quite messy:

```
import os
import csv
import numpy as np

hdir = '/Users/jswright/projects/MLSValidation/data/3.3/MLS/'
flst = [ f for f in os.listdir(hdir) if os.path.isfile(os.path.join(
    hdir,f)) ]
temp = []
for ii in range(len(flst)):
#-- append an empty dictionary for each profile
    temp.append({})
#-- slice the file name to store the date
    temp[ii]['year'] = int(flst[ii][0:4])
    temp[ii]['month'] = int(flst[ii][4:6])
    temp[ii]['day'] = int(flst[ii][6:8])
#-- get metadata
    with open(hdir+flst[ii]) as csvfile:
        reader = csv.reader(csvfile, delimiter=' ')
#------ distance is in the last position on the first line
        temp[ii]['mls3dist'] = float(filter(None,reader.next())[-1])
#------ longitude and latitude are on the second line
        row = np.array(filter(None,reader.next())).astype('float')
        temp[ii]['mls3lon'],temp[ii]['mls3lat'] = row
#------ time information is on the next line
        row = np.array(filter(None,reader.next())).astype('float')
        temp[ii]['mls3hour'],temp[ii]['mls3min'] = row[0:2]
#------ lists to hold the primary data
        temp[ii]['mls3prss'] = []
        temp[ii]['mls3o3mr'] = []
        temp[ii]['mls3snde'] = []
#------ loop through remaining lines and store data for validation
        for row in reader:
            row = np.array(filter(None,row)).astype('float')
            temp[ii]['mls3prss'].append(row[0])
            temp[ii]['mls3snde'].append(row[1])
            temp[ii]['mls3o3mr'].append(row[2])
```

In general, CSV files can be convenient for situations in which we have small data sets with limited metadata and variables that vary in only one dimension, but there are better file formats that we can use if we want to store large numbers of variables, data with multiple dimensions, or data with large amounts of associated metadata. We will encounter several of these as we go along.

Finally, it is worth noting that the `csv` module offers a convenient way to migrate from data analysis in Microsoft Excel and other spreadsheet-based programs to data analysis in python or other advanced computing languages. Excel spreadsheets can be saved as CSV files (File

> Save as > Comma Separated Values). Other spreadsheet programs have similar methods, generally under 'Save as' or 'Export'. We will learn easier methods for reading and writing csv files from python when we start working with the `pandas` module.