Numerical Integration

Carver J. Bierson

July 11, 2017

Like in root finding, not all integrals have analytic answers. There are two main cases we will look at here. The first is you have a function with a known form and the second is when you just have data points (or function evaluations).

As an example we may want to know the fraction of the Sun's energy radiated in the visible part of the solar spectrum. The amount of energy as a function of wavelength is given by

$$B_\lambda = \frac{2hc^2}{\lambda^5}\left(\frac{1}{e^{hc/\lambda kT}-1}\right) \tag{1}$$

where λ is the wavelength, h is Planck's constant, k is Boltzmann's constant, c is the speed of light, and T is the temperature. Stating our above question mathematically,

$$f_E = \frac{\int_{400nm}^{700nm} B_\lambda d\lambda}{\int_0^\infty B_\lambda d\lambda} \tag{2}$$

The bottom integral can be done analytically but the top one cannot.

1 Midpoint and Quadrature

The easiest to understand form of numerical integration is using the midpoint rule you may have learned in intro calculus. The idea is simply to evaluate our function at a series of points across the domain. This is also called the rectangle method as we are just summing the area of rectangles. We can write this mathematically as

$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i) \tag{3}$$

The more points we evaluate evaluate our function at (larger N) the better estimate we will get.

We can make this method one step better by summing trapezoids instead of rectangles. This method is called trapezoidal rule. There are two main functions that use this method. One is for when you have a function (*quad*) and the other if for when you have data(*trapz*).

```

import matplotlib.pyplot as plt
import numpy as np

```

```python

import scipy.optimize as opt
import scipy.integrate as integ

# define our function
def B_lmb(wavelength):
    # Plank function
    # Returns values in units of W/m^2 nm^-1 sr^-1
    # wavelength given in nm
    # Calculates the power power per wavelength
    h=6.626E-34 # Plank's constant (J/s)
    c=2.9979E8 # speed of light (m/s)
    k=  1.380065E-23 # Boltzmann's constant (J/K)
    T= 5777 # Temperature of the sun (K)
    wavelength=wavelength*1E-9
    return(1E-9*2*h*c**2/wavelength**5*
            (1.0/(np.exp(h*c/(wavelength*k*T))-1)))

# integrate
# the quad function retruns both the result and an estimate of the
    error
VisE, VisErr=integ.quad(B_lmb, 400, 700)

# we can use infinity as a limit
# Because this function peak sharply at small values I have split
    it in two
# Try doing it all at once to see the result

AllE, AllErr=integ.quad(B_lmb, 0, np.Inf)


print(VisE/AllE)
```

Note that here I had the user input the wavelength in nanometers and then convert it to meters. This has no effect when setting limits in quad however when integrating from $0 \leftrightarrow \infty$ quad is more stable if range of interest is between $1 - 10^4$ instead of $10^{-9} - 10^{-5}$.

The sun is well approximated by the Plank function, but that does not take into account the spectral features of the sun. We can try to estimate the same integrals using data of the suns spectrum.

```python

import numpy as np
import scipy.integrate as integ

filename='DataFiles/ASTMG173.csv'

Data=np.loadtxt(filename,skiprows=2,delimiter=',')

wvl= Data[:,0] # wavelength in nm
Energy = Data[:,1] # Energy at the Earth above the atmosphere (W/(m
    ^2 nm))


AllE = integ.trapz(Energy, x=wvl) # integrate over all wavelengths
    in file

wl_max=700 # nm
wl_min=400 # nm
Vis_index = np.logical_and(wvl>=wl_min, wvl<=wl_max) # Setup
    logical index for visible range

```

```
VisE = integ.trapz(Energy[Vis_index],x=wvl[Vis_index])



print(VisE/AllE)
```

These methods differ in their answers by ∼ 3% (check for yourself) but both agree that ∼40% of the sun's light is in the visible range.

2 Monte-Carlo

If you have a function, you can also estimate the integral of that function using a set of random numbers. To do this we do need to know the limits of integration, and the limits of the function on that domain. The general idea here is to randomly throw points within a square domain (in x and y) You can tally how many of those points fall below your curve and how many are above. This gives an estimate for the fraction of that domain that is occupied by your function It is easy to calculate that area of the box, and from that you get the area occupied by your function. Below is an example

```

import matplotlib.pyplot as plt
import numpy as np

import numpy.random as rnd

import scipy.optimize as opt
import scipy.integrate as integ

import time # for checking speed

# define our function
def B_lmb(wavelength):
    # Plank function
    # Returns values in units of W/m^2 nm^-1 sr^-1
    # wavelength given in nm
    # Calculates the power power per wavelength
    h=6.626E-34 # Plank's constant (J/s)
    c=2.9979E8 # speed of light (m/s)
    k=  1.380065E-23 # Boltzmann's constant (J/K)
    T= 5777 # Temperature of the sun (K)
    wavelength=wavelength*1E-9
    return(1E-9*2*h*c**2/wavelength**5*
            (1.0/(np.exp(h*c/(wavelength*k*T))-1)))



# plot up function
Plotx=np.linspace(0,5000,100)

plt.figure()
plt.plot(Plotx, B_lmb(Plotx), 'k')

Ndarts=int(1E4) # number of mote-carlo darts to throw for
    integration
xlimits=[400.0,700.0] # The domain we want to integrate over
ylimits=[0.0,40.0E3] # the maximum of the function over this range
    is ~26E3

```

```python
TotalArea = (xlimits[1]-xlimits[0])*(ylimits[1]-ylimits[0])

# I will use a loop method and vector method and check which is faster

Lt0 = time.time()
# Loop method
countbelow=0
for i in range(Ndarts):
    # Generate a random x-y point in the range above
    x= rnd.rand()*(xlimits[1]-xlimits[0])+xlimits[0]
    y= rnd.rand()*(ylimits[1]-ylimits[0])+ylimits[0]

    if y<B_lmb(x): # if it is below our function
        countbelow+=1 # count it
        plt.plot(x,y,'b.') # plot it blue
    else:
        plt.plot(x,y,'r.') #otherwise plot it red
        #pass # if you comment out the plot you need pass to have something here

IntArea=float(countbelow)/float(Ndarts)*TotalArea # float function keeps integer math from messing this up
Lt1 = time.time()

#Vector method
# make arrays of random x-y points
Vt0 = time.time()
xarr= rnd.rand(Ndarts)*(xlimits[1]-xlimits[0])+xlimits[0]
yarr= rnd.rand(Ndarts)*(ylimits[1]-ylimits[0])+ylimits[0]

BelowIndex=yarr<B_lmb(xarr) #create a logical array for who is below the curve

countbelow2=np.sum(BelowIndex) # Trues count as 1, False is 0



IntArea2=float(countbelow2)/float(Ndarts)*TotalArea # float function keeps integer math from messing this up

#make new figure
plt.figure()
plt.plot(Plotx, B_lmb(Plotx), 'k')

plt.plot(xarr[BelowIndex],yarr[BelowIndex],'b.') # plot it blue if below
plt.plot(xarr[np.logical_not(BelowIndex)],yarr[np.logical_not(BelowIndex)],'r.') #otherwise plot it red

Vt1=time.time()

plt.xlabel('Wavelength (nm)')
plt.xlabel('Energy ( W/m^2 nm^-1 sr^-1)')

# integrate with quad for comparison
# the quad function retruns both the result and an estimate of the error
VisE, VisErr=integ.quad(B_lmb, 400, 700)

# Compare answers
print('Quad\tMonte Carlo 1\tMonte Carlo 2')
print('{:0.3E}\t{:0.3E}\t{:0.3E}'.format(VisE,IntArea,IntArea2))
print('')
print('Speed check (s)')
print('Loop\tVector\tDifference')
Ldt=Lt1-Lt0
Vdt=Vt1-Vt0
print('{:0.3f}\t{:0.3f}\t{:0.3f}'.format(Ldt,Vdt,Ldt-Vdt))
```

In the above example I do the Monte-Carlo integration as a loop and as a vector process. In my test I found the vector method was **ninety times faster**. I commented out all the plotting commands to try and be fair to the loop in my test.