Computational Physics HW1

Luke Bouma

July 23, 2015

1 Another ball dropped from a tower

A ball is dropped from a tower of height h with initial velocity zero. Write a program that asks the user to enter the height in meters of the tower, and then calculates and prints the time the ball takes until it hits the ground, ignoring air resistance. Use this program to find the time for a ball dropped from a 100m high tower.

The ball falls as

$$y(t) = h - \frac{1}{2}gt^2 \tag{1}$$

which means that it hits the ground at $t = \sqrt{2h/g}$. The code we write to compute the time the ball takes to hit the ground is then

```
from math import sqrt
g = 9.81
h = float(input("Tower height (m): "))
print("Time to hit ground is",sqrt(2*h/g),"seconds")
```

which gives us $\boxed{\text{4.52 seconds}}$ for an initial height of 100m.

2 Quantum potential step

Electron, mass $m = 9.11 \times 10^{-31}$kg, encounters a one-dimensional potential step. It has initial kinetic energy $E = 10$eV, and wavevector $k_1 = \sqrt{2mE}/\hbar$. The jump in potential energy is of height $V = 9$eV, at $x = 0$. When $E > V$, solving the Schrödinger equation yields the result that the particle either (a) is transmitted, with kinetic energy $E - V$ and wavevector $k_2 = \sqrt{2m(E-V)}/\hbar$, or (b) is reflected, with all its kinetic energy and keeping k_1. The probabilities for transmission and reflection are

$$T = \frac{4k_1k_2}{(k_1+k_2)^2}, \quad R = \left(\frac{k_1-k_2}{k_1+k_2}\right)^2. \tag{2}$$

We want a program that computes and prints the transmission and reflection probabilities using Eq. (2).

Clearly the interesting part about this is the units. $\hbar = 1.05 \times 10^{-34}$Js is 16 orders of magnitude different than the 10eV $= 1.602 \times 10^{-18}$J, ditto the electron mass. So set $m = \hbar = 1$. The calculation becomes

$$
\begin{aligned}
T &= \frac{4k_1k_2}{(k_1+k_2)^2} & (3)\\
&= \frac{4m}{\hbar^2}\times\frac{\hbar^2}{m}\times\frac{\sqrt{E}\sqrt{(E-V)}}{(\sqrt{E}+\sqrt{(E-V)})^2} & (4)
\end{aligned}
$$

which means that the units cancel, as we would expect for probabilities. We leave in placeholder values in the code we write:

```
from math import sqrt

E = 10.     #eV
V = 9.      #eV
hbar = 1.
m = 1.

k1 = sqrt(2*m*E)/hbar
k2 = sqrt(2*m*(E-V))/hbar

T = 4*k1*k2/(k1+k2)**2
R = ((k1-k2)/(k1+k2))**2

print("Transmission:",T,"Reflection:",R)
```

which prints $T = 0.7301$, $R = 0.2699$.

3 Madelung constant

Madelung constant gives total electric potential felt by an atom in a solid – it depends on the charges and the other nearby atoms. For NaCl there are alternating Na^+ and Cl^- ions. Label each position on the lattice by (i, j, k) integers, and let Na atoms fall where $i+j+k$ is even, and Cl fall where the sum is odd.

Consider a Na at $(0, 0, 0)$, and calculate the Madelung constant. If the lattice spacing is a, then the distance from the origin to the atom at position (i, j, k) is

$$d = a\sqrt{i^2+j^2+k^2}$$

and the potential at the origin created by such an atom is

$$V(i,j,k) = \pm\frac{e}{4\pi\epsilon_0 a\sqrt{i^2+j^2+k^2}},$$

for ϵ_0 the permittivity of the vacuum, and the sign of the expression determining whether $i+j+k$ is even or odd. The total potential felt by the sodium atom is then the sum of this quantity over all other atoms. Let us assume a cubic box around the sodium atom at the origin, with L atoms in all directions. Then

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L\\ \text{not all }=0}}^{L} V(i,j,k) = \frac{e}{4\pi\epsilon_0 a}M$$

where M is (approximately) the Madelung constant for large L. Write a program to calculate and print the Madelung constant for NaCl. The program we write follows:

```
from math import sqrt, pi
from numpy import empty
e = 1.602e-19              #electron charge, C
a = 5.6402e-10             #lattice constant, m
eps0 = 8.854187e-12        #vacuum permittivity, SI
prefac = e/(4*pi*eps0*a)

L = 150

V = empty([L,L,L],float)
Vtot = 0.

for i in range(-L,L,1): #n.b. this goes one-too-far on one end
    for j in range(-L,L,1):
        for k in range(-L,L,1):
            denom = sqrt(i**2+j**2+k**2)
            if not(i==j==k==0):
                V[i,j,k] = prefac/denom
            if ((i+j+k)%2 == 0) and not(i==j==k==0):
                Vtot += V[i,j,k]
            elif ((i+j+k)%2 == 1) and not(i==j==k==0):
                Vtot -= V[i,j,k]

print(Vtot/prefac)
```

which gives $M = \pm 1.74756$ depending on if the charge at the origin is positive (Na, so negative constant) or negative (then positive constant). This matches the literature value from Wiki.

4 The semi-empirical mass formula

The semi-empirical mass formula is a formula for calculating the approximate nuclear binding energy B of an atomic nucleus with atomic number Z and mass number A:

$$B = a_1 A - a_2 A^{2/3} - a_3 \frac{Z^2}{A^{1/3}} - a_4 \frac{(A-2Z)^2}{A} + \frac{a_5}{A^{1/2}}$$

where, in units of MeV, the constants are $a_1 = 15.67$, $a_2 = 17.23, a_3 = 0.75$, and $a_4 = 93.2$. Finally,

$$a_5 = \begin{cases} 0 & \text{if } A \text{ is odd,} \\ 12.0 & \text{if } A \text{ and } Z \text{ are both even,} \\ -12.0 & \text{if } A \text{ is even and } Z \text{ is odd.} \end{cases}$$

4.1 Program that takes A, Z and outputs binding energy

```
from numpy import sqrt

A = 58
Z = 28
a1 = 15.67
```

```
a2 = 17.23
a3 = 0.75
a4 = 93.2
a5 = 0.
if A % 2 == 1:
    a5 = 0.
elif A % 2 == 0 and Z % 2 == 0:
    a5 = 12.
elif A % 2 == 0 and Z % 2 == 1:
    a5 = -12.

B = a1*A - a2*A**(2/3) - a3*Z**2/A**(1/3) \
    -a4*(A-2*Z)**2/A + a5/sqrt(A)

print(B)                    #prints 494MeV
```

4.2 Modification

Yields $B/A = 8.516$MeV/nucleon.

4.3 Find most stable nucleus

The program below gives mass number 58 binding energy 8.516 MeV for $Z = 28$.

```
from numpy import sqrt

Z = 28
a1 = 15.67
a2 = 17.23
a3 = 0.75
a4 = 93.2
a5 = 0.
mostStable = 0.

for A in range(Z,3*Z,1):
    if A % 2 == 1:
        a5 = 0.
    elif A % 2 == 0 and Z % 2 == 0:
        a5 = 12.
    elif A % 2 == 0 and Z % 2 == 1:
        a5 = -12.
    B = a1*A - a2*A**(2/3) - a3*Z**2/A**(1/3) \
        -a4*(A-2*Z)**2/A + a5/sqrt(A)
    if B/A > mostStable:
        mostStable = B/A
        stableA, stableB = A, B

print("mass number",stableA, "binding energy",stableB/stableA,"MeV")
```

4.4 Running through all Z up to 100, all A too.

With the program modified to run through all the atomic numbers, and all mass numbers from the atomic number to 3 times the atomic number, we get the result: Z: 24 stable A: 50 B/A: 8.533 MeV/nucleon. This is, as noted, different from the real life answer (by four protons!), but I'm pretty sure the code is fine:

```
from numpy import sqrt

a1 = 15.67
a2 = 17.23
a3 = 0.75
a4 = 93.2
a5 = 0.
mostStable = -5e5

for Z in range(1,101,1):
    for A in range(Z,3*Z+1,1):
        if A % 2 == 1:
            a5 = 0.
        elif A % 2 == 0 and Z % 2 == 0:
            a5 = 12.
        elif A % 2 == 0 and Z % 2 == 1:
            a5 = -12.
        B = a1*A - a2*A**(2/3) - a3*Z**2/A**(1/3) \
            -a4*(A-2*Z)**2/A + a5/sqrt(A)
        if B/A > mostStable:
            mostStable = B/A
            stableA, stableB = A, B
    print("Z:",Z, "stable A:",stableA, "B/A:",stableB/stableA,"MeV/nucleon")
```

5 Binomial Coefficients

The binomial coefficient $\binom{n}{k}$ is an integer equal to

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times (n-2) \times ... \times (n-k+1)}{1 \times 2 \times ... \times k}$$

when $k \geq 1$, or $\binom{n}{0} = 1$ when $k = 0$.

5.1 Write binomial(n,k) to calculate binomial coefficient (integer)

```
def fac(x):
    if x == 1:
        return 1
    else:
        return x*fac(x-1)
```

```
def binomial(n,k):
    if k == 0 or n==k:
        return 1
    else:
        return fac(n)//(fac(k)*(fac(n-k)))
```

These functions calculate the binomial coefficient for a given n and k. They return the answer in terms of an integer, and give the correct value of 1 for $k = 0$.

5.2 Print 20 lines of Pascal's triangle

```
def fac(x):
    if x == 1:
        return 1
    else:
        return x*fac(x-1)

def binomial(n,k):
    if k == 0 or n==k:
        return 1
    else:
        return fac(n)//(fac(k)*(fac(n-k)))

for n in range(1,21):
    for k in range(0,n+1):
        print(binomial(n,k),end=" ")
    print("\n")
```

```
gives

1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1
1 16 120 560 1820 4368 8008 11440 12870 11440 8008 4368 1820 560 120 16 1
1 17 136 680 2380 6188 12376 19448 24310 24310 19448 12376 6188 2380 680 136 17
1 18 153 816 3060 8568 18564 31824 43758 48620 43758 31824 18564 8568 3060 816 1
1 19 171 969 3876 11628 27132 50388 75582 92378 92378 75582 50388 27132 11628 38
```

5.3 Coin flipping probability

```
n = 100
k = 60
print("(a): prob is", binomial(n,k)/2**n)

probSum = 0
for j in range(k,101):
    probSum += binomial(n,j)/2**n
print("(b): prob is", probSum)
```

gives the output

```
(a): prob is 0.010843866711637987
(b): prob is 0.028443966820490392
```

6 Prime numbers

Observe that: (a) a number n is prime if it has no prime factors less than n. Hence we only need to check if it is divisible by other primes. (b) If n is non-prime with factor r, then $n = rs$, where s is also a factor. If $r \geq \sqrt{n}$ then $n = rs \geq \sqrt{n}s$ which implies that $s \leq \sqrt{n}$. Thus to check if a number is prime we have to check it prime factors only up to and including $\sqrt{n}$– if there are none then number is prime. (c) If we find a single prime factor less than $\sqrt{n}$ then we know the number is non-prime, and hence there is no need to check any further – we can abandon this number and move on to something else.

With these observations, write a program that finds all the primes up to 10,000. Make a list to store them, which starts with just 2 in it. Then for each number n from 3 to 10,000 check whether to number is divisible by any of the primes in the list up to and including $\sqrt{n}$. As soon as you find a single prime factor you can stop checking the rest of them – you know n is not a prime. If you find no prime factors $\sqrt{n}$ or less then n is prime and you should add it to the list. Then print out the list all in one go at the end of the program.

```
from numpy import size, sqrt, ceil
primes = [2]
for n in range(3,10000):
    primeLen = size(primes)
    rootN = ceil(sqrt(n))
    for k in range(0,primeLen):
        if n % primes[k] == 0:
            break
        if primes[k] >= rootN:
            primes.append(n)
            break

for i in range(0, size(primes)):
    print(primes[i])
```

7 Recursion

7.1 Catalan numbers

The Catalan numbers C_n are defined as

$$C_n = \begin{cases} 1 & \text{if } n = 0 \\ \frac{4n-2}{n+1} C_{n-1} & \text{if } n > 0 \end{cases}$$

Write a Python function, using recursion, to calculate C_n. Use this function to calculate C_{100}.

```
def catalan(n):
    if n == 0:
        return 1
    elif n > 0:
        return (4*n-2)/(n+1) * catalan(n-1)

print(catalan(100))                 #gives 8.96519947...e+56
```

7.2 Great common divisor

Euclid showed that the greatest common divisor $g(m, n)$ of two nonnegative integers m and n satisfies

$$g(m, n) = \begin{cases} m & \text{if } n = 0, \\ g(n, m \bmod n) & \text{if } n > 0. \end{cases}$$

Write a Python function `g(m,n)` that employs recursion to find the gcd of 108 and 192 using this formula.

```
def g(m,n):
    if n == 0:
        return m
    elif n > 0:
        return g(n, m%n)

print(g(108,192))
```

which prints the result 12.