CSC 431 - Scientific Computing

Midterm By Zhenyang Lu

PART I

Problem 1

Implement a Taylor expansion for log(1+x) using Python or C++. Discuss convergence and reminder within the convergence radius. No need to prove that it does not converge outside known convergence radius.

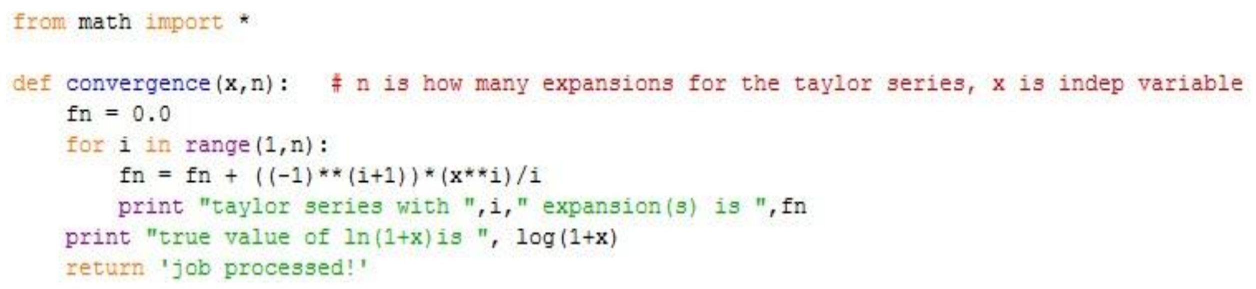
Taylor Series for this formula, at x=0, is

$$\mathrm{Ln}(1+x) = \sum_{i=1}^{\infty} (-1)^{i+1} X^{i}/i + O(X^{i+1})$$

Converges when |x| <1.

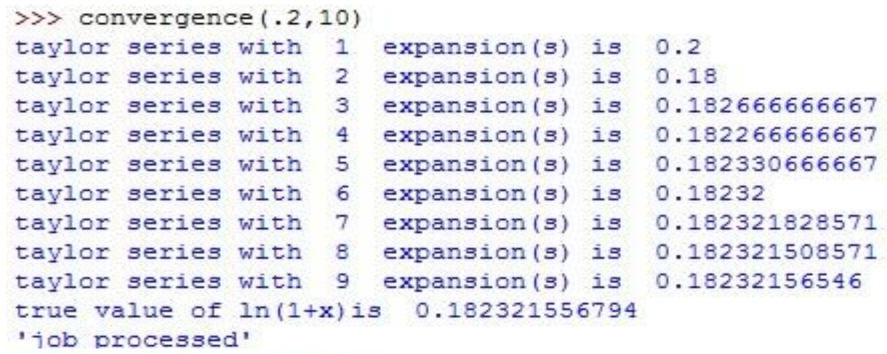
(1). Convergence

To prove it converges, I will run Python script like:

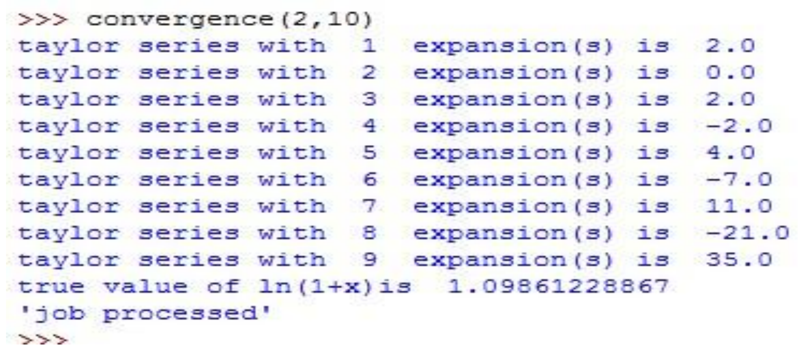


```
from math import *

def convergence(x,n):   # n is how many expansions for the taylor series, x is indep variable
    fn = 0.0
    for i in range(1,n):
        fn = fn + ((-1)**(i+1))*(x**i)/i
        print "taylor series with ",i," expansion(s) is ",fn
    print "true value of ln(1+x)is ", log(1+x)
    return 'job processed!'
```

The result shows that it converges for |x| <1, while digresses when |x|>1.



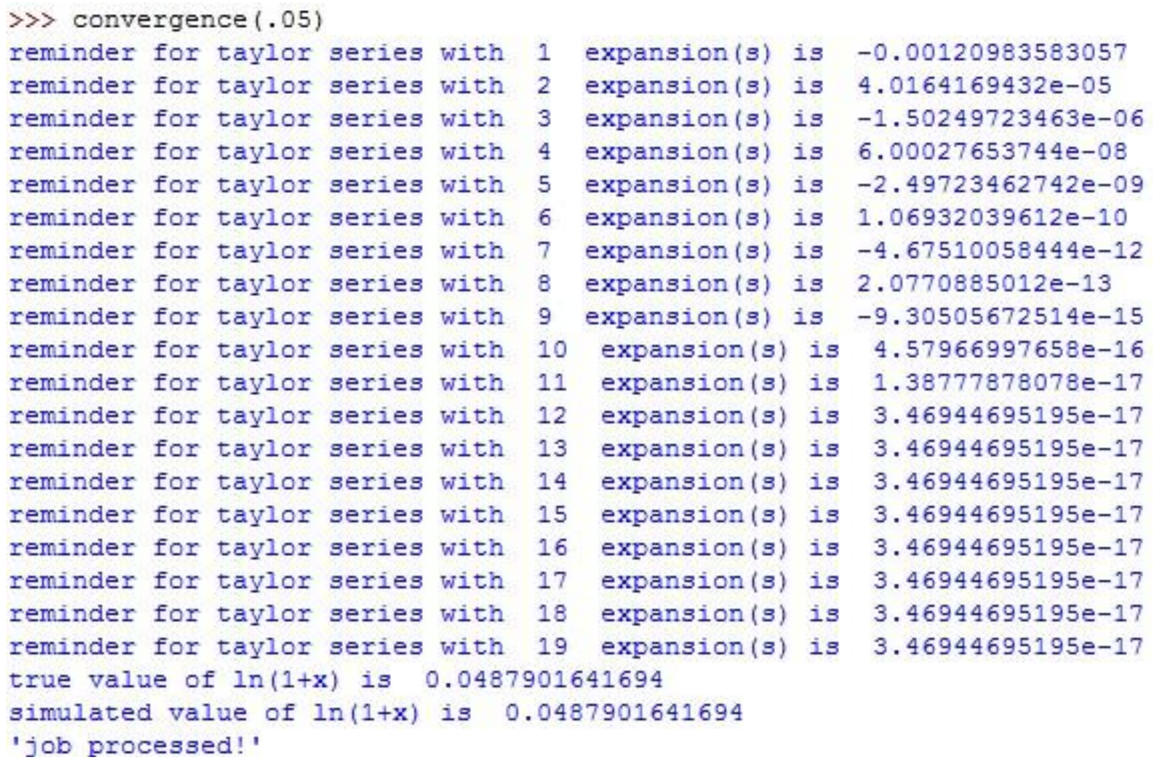
```
>>> convergence(.2,10)
taylor series with  1  expansion(s) is  0.2
taylor series with  2  expansion(s) is  0.18
taylor series with  3  expansion(s) is  0.182666666667
taylor series with  4  expansion(s) is  0.182266666667
taylor series with  5  expansion(s) is  0.182330666667
taylor series with  6  expansion(s) is  0.18232
taylor series with  7  expansion(s) is  0.182321828571
taylor series with  8  expansion(s) is  0.182321508571
taylor series with  9  expansion(s) is  0.18232156546
true value of ln(1+x)is  0.182321556794
'job processed'
```



```
>>> convergence(2,10)
taylor series with  1  expansion(s) is  2.0
taylor series with  2  expansion(s) is  0.0
taylor series with  3  expansion(s) is  2.0
taylor series with  4  expansion(s) is  -2.0
taylor series with  5  expansion(s) is  4.0
taylor series with  6  expansion(s) is  -7.0
taylor series with  7  expansion(s) is  11.0
taylor series with  8  expansion(s) is  -21.0
taylor series with  9  expansion(s) is  35.0
true value of ln(1+x)is  1.09861228867
'job processed'
>>>
```

(2) Reminder

By slightly change the script, we could obtain the **conclusion that the reminder goes to zero as the taylor series become more and more precise.** For example, below is the result of x= 0.05, we could see that the reminder is almost zero after reaching certain precision. Actually, the simulated value and true value computing by Python library is the same.



```
>>> convergence(.05)
reminder for taylor series with  1  expansion(s) is  -0.00120983583057
reminder for taylor series with  2  expansion(s) is  4.0164169432e-05
reminder for taylor series with  3  expansion(s) is  -1.50249723463e-06
reminder for taylor series with  4  expansion(s) is  6.00027653744e-08
reminder for taylor series with  5  expansion(s) is  -2.49723462742e-09
reminder for taylor series with  6  expansion(s) is  1.06932039612e-10
reminder for taylor series with  7  expansion(s) is  -4.67510058444e-12
reminder for taylor series with  8  expansion(s) is  2.0770885012e-13
reminder for taylor series with  9  expansion(s) is  -9.30505672514e-15
reminder for taylor series with  10  expansion(s) is  4.57966997658e-16
reminder for taylor series with  11  expansion(s) is  1.38777878078e-17
reminder for taylor series with  12  expansion(s) is  3.46944695195e-17
reminder for taylor series with  13  expansion(s) is  3.46944695195e-17
reminder for taylor series with  14  expansion(s) is  3.46944695195e-17
reminder for taylor series with  15  expansion(s) is  3.46944695195e-17
reminder for taylor series with  16  expansion(s) is  3.46944695195e-17
reminder for taylor series with  17  expansion(s) is  3.46944695195e-17
reminder for taylor series with  18  expansion(s) is  3.46944695195e-17
reminder for taylor series with  19  expansion(s) is  3.46944695195e-17
true value of ln(1+x) is  0.0487901641694
simulated value of ln(1+x) is  0.0487901641694
'job processed!'
```

Problem 2

Explain why a divergent infinite series such as $\sum_{n=1}^{\infty} 1/N$, can have a finite sum in floating point arithmetic. At what point will the partial sum cease to change?

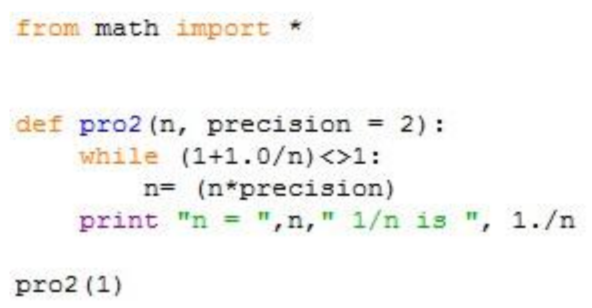
(1) EXPLAIN

Since as N goes larger, 1/N reaches 0 (but in math, never touch zero), since the length of floating number that a computer can present is limited, when it reaches the limit, 1/n will be regarded as 0. Thus, although in math there should be infinite series that be summed up, in practice the computer deems it as finite series that summed up since the floatation algorithm can not present such a small number and instead be treated as zero!

(2) CEASING POINT

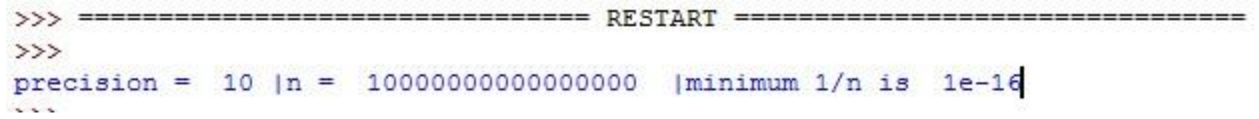
Due to the speed of the computer, I can not run a python program to look for the cutting point that makes the summation to cease growing by intuitive way, which means to compute sum of 1/N until it stops. However, I think that since the second derivative of function f(x) = 1/x is always positive, meaning that it is always growing (but far less likely to be overflow than for 1/N to be underflow in floatation algorithm) in math theory, so **this formula, $\sum_{n=1}^{\infty} 1/N$, is converged in python!** In that way, I think what we have to do to solve the problem is to find the smallest 1/N when N goes to infinity. That N would be the cutting point for the summation to cease growing.

The script to find smallest 1/N is like



```
from math import *


def pro2(n, precision = 2):
    while (1+1.0/n)<>1:
        n= (n*precision)
    print "n = ",n," 1/n is ", 1./n

pro2(1)
```

To speed up my program, I do not compute like 1/N while N = 1,2,3….infinity, I assume N = 1,2,4,6,12….2*N, or N = 1,10,100,1000…..10*N, which is represented by the parameter called *precision* in my script. For stop loop condition, it is when 1+1.0/N is not different from 1 to Python. The result that gets the smallest 1/N happens when precision is 10, it goes like:



```
>>> ================================ RESTART ================================
>>>
precision =  10 |n =  10000000000000000  |minimum 1/n is  1e-16
```

That is, when N is around 10^{16}, Python (64-bits) assumes that the formula $\sum_{n=1}^{\infty} 1/N$ stop growing! Notice that it is may be not a precise figure.

Problem 3

Consider the expression:

$$\frac{\mathbf{1}}{\mathbf{1-x}}-\frac{\mathbf{1}}{\mathbf{1+x}}$$

For what range of values of x it is difficult to compute this expression accurately in floating point arithmetic? Provide an alternative form for the same expression which gives more accurate results.

Intuitively, If $x \to 1, then\ \frac{1}{1-x} \to +\infty$, If $x \to -1, then\ \frac{1}{1+x} \to +\infty$. In that way, the formula becomes tough to solve. By reforming the formula, we get

$$\frac{\mathbf{1}}{\mathbf{1-x}}-\frac{\mathbf{1}}{\mathbf{1+x}}=\frac{\mathbf{1+x}}{(\mathbf{1-x})(\mathbf{1+x})}-\frac{\mathbf{1-x}}{(\mathbf{1-x})(\mathbf{1+x})}=\mathbf{2x}/(\mathbf{1-x^2})$$

However, I found a very strange result, by computing the original ploy by hand, I got the result is 9999.4999749987498499374688, apparently, the original ploy is better in precision.

Sum, I do not know which one is better, maybe sometimes the original is the better, the other sometimes is better

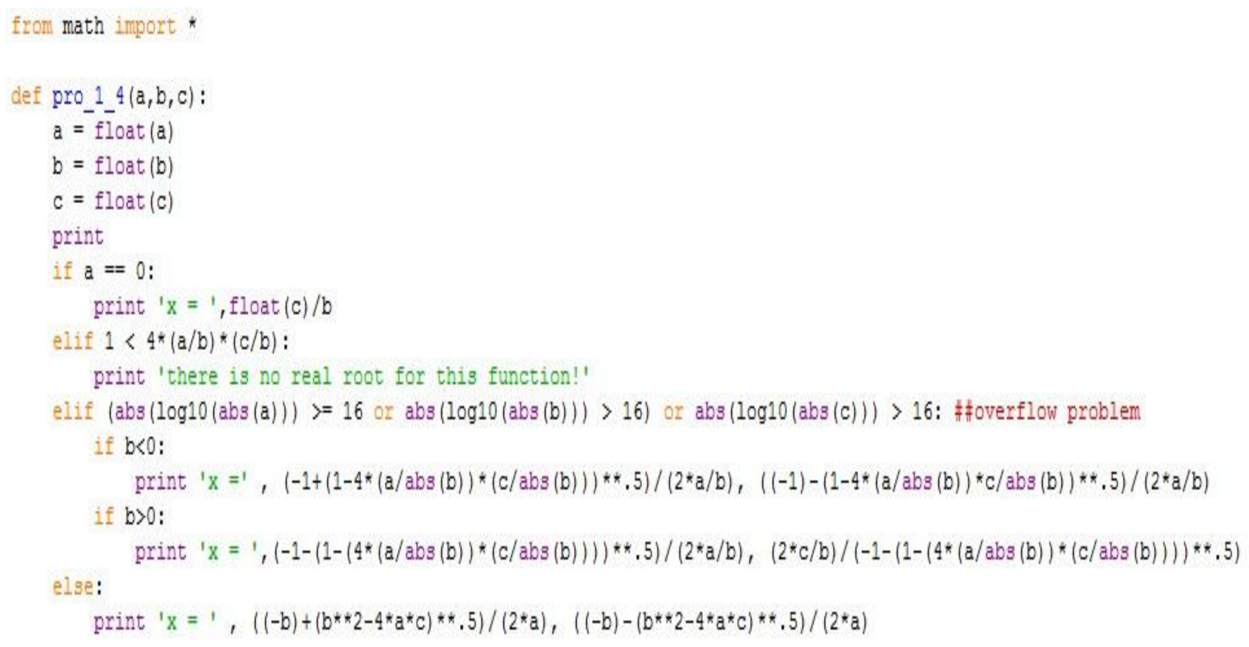
Problem 4

Write a program in Python or C++ to solve the quadratic equation $ax^2+bx+c=0$ using the standard quadratic formula $x=(-b\pm\sqrt{b^2-4ac})/(2a)$ or the alternate formula $x=(2c)/(-b\pm\sqrt{b^2-4ac})$ You should detect complex routes but no need to solve for complex values.

For my script, I will first check if a=0. If it is so, thing becomes easier since it is linear equation.

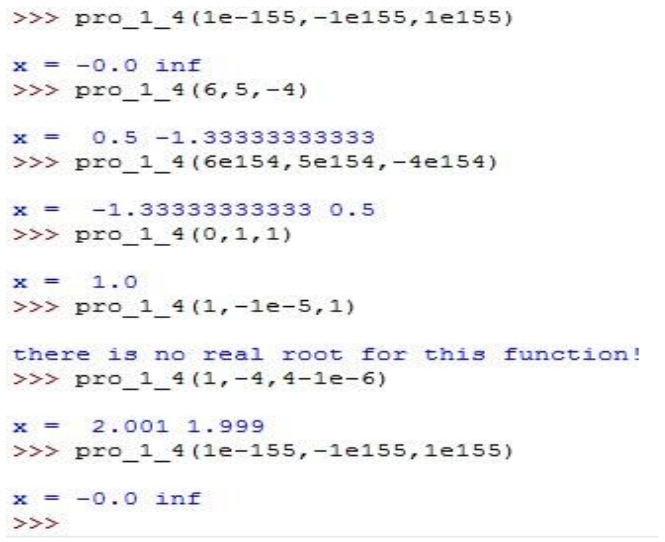
Second, I will check if b^2-4ac <0, if it is the case, there will be no real root for the equation.

Third, I will check if a or b or c themselves are overflow or underflow, for my Python the smallest number and largest number that can be present in Python are 1e-16 and 1e16, so if a or b or c flow over that bounds, I will rescale them by divide one of them. However, for the six pair of parameters, that is, a= 1e-155, b= -1e155,c=1e155, the solution is either 0 or infinity, which is a special case. The code for the program is like:



```
from math import *

def pro_1_4(a,b,c):
    a = float(a)
    b = float(b)
    c = float(c)
    print
    if a == 0:
        print 'x = ',float(c)/b
    elif 1 < 4*(a/b)*(c/b):
        print 'there is no real root for this function!'
    elif (abs(log10(abs(a))) >= 16 or abs(log10(abs(b))) > 16) or abs(log10(abs(c))) > 16: ##overflow problem
        if b<0:
            print 'x =' , (-1+(1-4*(a/abs(b))*(c/abs(b)))**.5)/(2*a/b), ((-1)-(1-4*(a/abs(b))*c/abs(b))**.5)/(2*a/b)
        if b>0:
            print 'x = ',(-1-(1-(4*(a/abs(b))*(c/abs(b))))**.5)/(2*a/b), (2*c/b)/(-1-(1-(4*(a/abs(b))*(c/abs(b))))**.5)
    else:
        print 'x = ' , ((-b)+(b**2-4*a*c)**.5)/(2*a), ((-b)-(b**2-4*a*c)**.5)/(2*a)
```

Running of the code is like:



```
>>> pro_1_4(1e-155,-1e155,1e155)

x = -0.0 inf
>>> pro_1_4(6,5,-4)

x =  0.5 -1.33333333333
>>> pro_1_4(6e154,5e154,-4e154)

x =  -1.33333333333 0.5
>>> pro_1_4(0,1,1)

x =  1.0
>>> pro_1_4(1,-1e-5,1)

there is no real root for this function!
>>> pro_1_4(1,-4,4-1e-6)

x =  2.001 1.999
>>> pro_1_4(1e-155,-1e155,1e155)

x = -0.0 inf
>>>
```

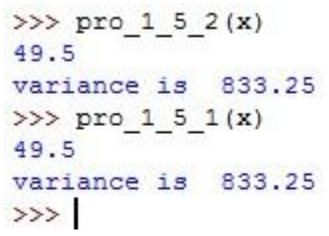
Problem 5

Write a program in Python or C++ to compute the variance of a sequence {x*i*} using the two equivalent formulas:

$$\sigma^2 = 1/n \sum_{i=0}^{i<n}(x_i - \bar{x})^2 = 1/n(\sum_{i=0}^{i<n} x_i^2 - n\bar{x}^2)$$

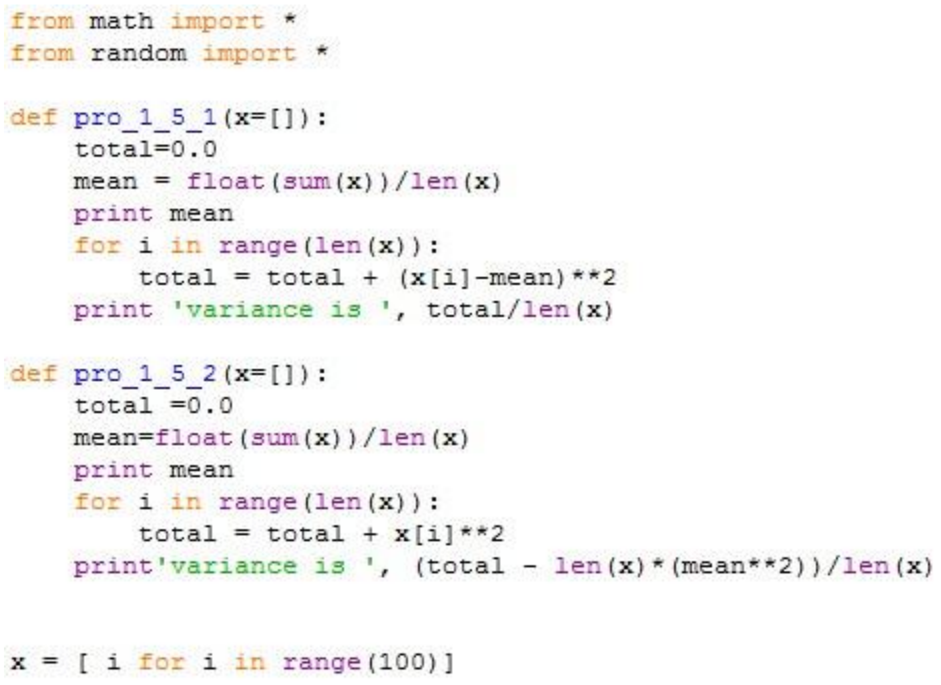
Prove that the two expressions are equivalent. Conceive an input series {x*i*} for which the two expressions produce different results numerically. Is one better than the other?

By creating a sequence that contains from 1 to 100, we could run a program on that and the result shows those two expressions are the same.



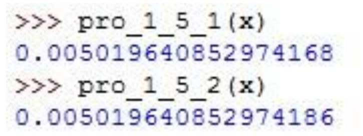
```
>>> pro_1_5_2(x)
49.5
variance is  833.25
>>> pro_1_5_1(x)
49.5
variance is  833.25
>>> 
```

The code I use:



```
from math import *
from random import *

def pro_1_5_1(x=[]):
    total=0.0
    mean = float(sum(x))/len(x)
    print mean
    for i in range(len(x)):
        total = total + (x[i]-mean)**2
    print 'variance is ', total/len(x)

def pro_1_5_2(x=[]):
    total =0.0
    mean=float(sum(x))/len(x)
    print mean
    for i in range(len(x)):
        total = total + x[i]**2
    print'variance is ', (total - len(x)*(mean**2))/len(x)


x = [ i for i in range(100)]
```

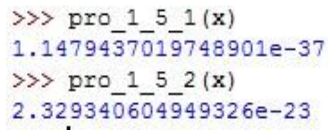
However, I could come up with two sequences that have different computing result by these two formulas.
First, if I create a sequence like X = [1, 0.1, 0.01…..1/10^n], which is a geometric sequence. Then the results of those two formulas are slightly different.



```
>>> pro_1_5_1(x)
0.005019640852974168
>>> pro_1_5_2(x)
0.005019640852974186
```

Notice that two results are slightly different.

If we build a special sequence like X= [0.1, 0.1, ……0.1] or even X=[1/10^n, 1/10^n….1/10^n], n =1,2,3,4……k. Then we could also get very different results by those two variance formula. Below I present the results on a sequence that [1/10000, 1/10000, 1/10000……1/10000], which has 200 elements. The Python says:



```
>>> pro_1_5_1(x)
1.1479437019748901e-37
>>> pro_1_5_2(x)
2.329340604949326e-23
```

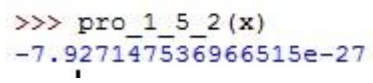
I prefer the original formula, which is

$$\boldsymbol{\sigma^2 = 1/n \sum_{i=0}^{i<n} (x_i - \bar{x})^2}$$

Since in the second one, the sum of x^2 is computed alone, if x^2 is small enough, the second formula,

$$\boldsymbol{\left(\sum_{i=0}^{i<n} {x_i}^2 - n\bar{x}^2\right)}$$

is likely to be negative, which is unlikely for a variance to be. My program has also shown this conclusion, when the sequence is [1/1000000, 1/1000000……..1/1000000] with 300 elements.



```
>>> pro_1_5_2(x)
-7.927147536966515e-27
```

Problem 6

The function returns the Black-Scholes price for a European call option. S and X are dollar amounts. Will the result be more accurate or less accurate if S and X are expressed in cents? What if they are expressed in thousands of dollars?

Will the result be more accurate or less accurate if time is pressed in years instead of days (and accordingly r, σ are yearly return and volatility)?

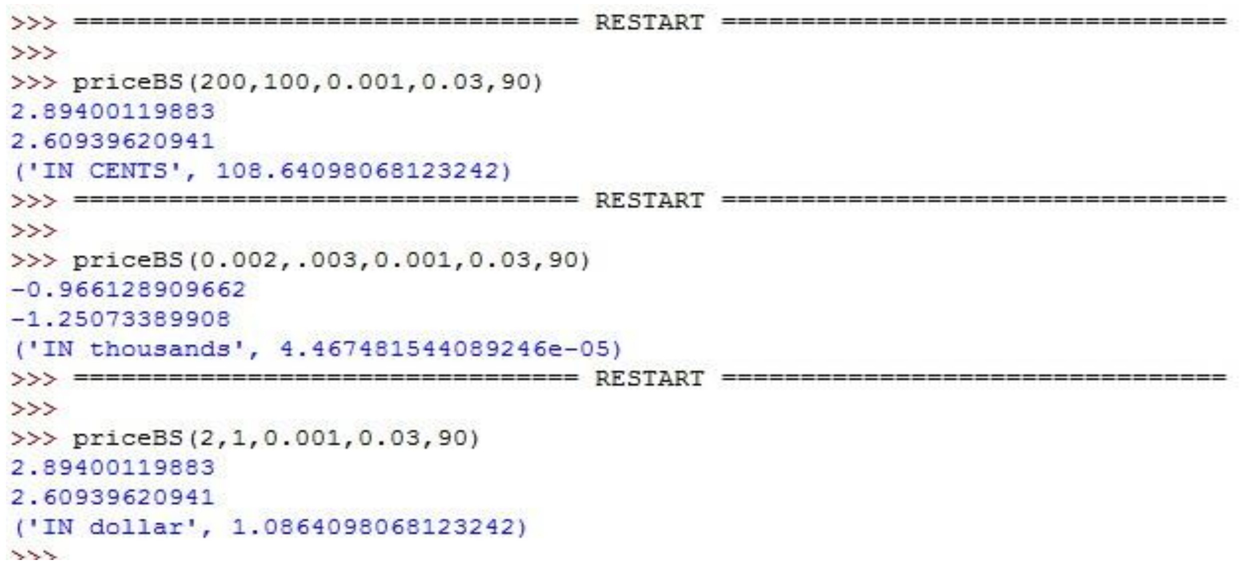
Plot the output for X = 10:0; r = 0:001; σ= 0:03; t = 90:0 as function of S in range [0; 20].

(1) Cents or Thousands dollars?

Since it is a call option, it will never happen that S<X, saying the call option is worth zero. Also, given that stock option will seldom quote a price like million dollars per contract, I do not think overflow will happen in this model, so underflow will be the potential problem for us.

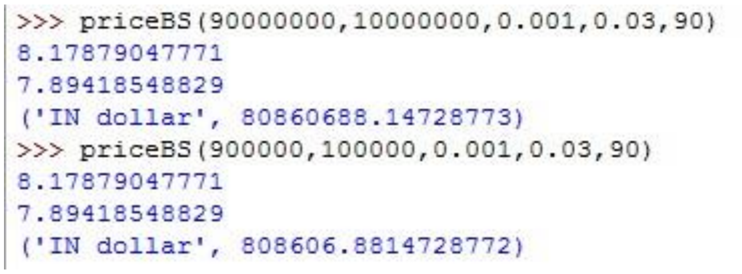
Intuitively, by looking at the equation to get priceBS, X and S only matter in these equation, c = S*cdf(d1)-X*exp(-r*t)*cdf(d2) and d1 = (log(S/X)+r*t)/(sigma*sqrt_t)+0.5*sigma*sqrt_t.

Assume X = 1, t = 90, r= 0.001, σ= 0:03, S = 2 in Dollar amount, that is 100 and 200 cents and 0.001 and 0.002 thousand dollars. By running the Python script, we see that



```
>>> ================================ RESTART ================================
>>>
>>> priceBS(200,100,0.001,0.03,90)
2.89400119883
2.60939620941
('IN CENTS', 108.64098068123242)
>>> ================================ RESTART ================================
>>>
>>> priceBS(0.002,.003,0.001,0.03,90)
-0.966128909662
-1.25073389908
('IN thousands', 4.467481544089246e-05)
>>> ================================ RESTART ================================
>>>
>>> priceBS(2,1,0.001,0.03,90)
2.89400119883
2.60939620941
('IN dollar', 1.0864098068123242)
>>>
```

Apparently, price seems not to influence so much on it. The second result is negative since X> S which in real business means that the price of option should be zero! Below is another example in the situation that the figure is very large!



```
>>> priceBS(90000000,10000000,0.001,0.03,90)
8.17879047771
7.89418548829
('IN dollar', 80860688.14728773)
>>> priceBS(900000,100000,0.001,0.03,90)
8.17879047771
7.89418548829
('IN dollar', 808606.8814728772)
```

From that I think that there is no big difference. Although there may be difference in digits that are very far away from the floating point, I do not think it is a problem since in real business that could never happed that the price is so small.

(2) Yearly T or Daily T?

If it is yearly, ***$r = r_{daily}$*360; $\sigma = \sigma_{daily}$ *360**.5 and $t = t_{daily}$/360***.

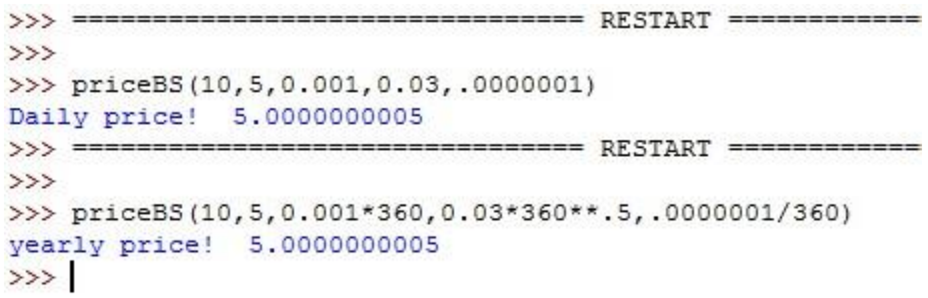
For our test, I assume yearly parameters: X=10, S=5, r =0.36, σ = 0.03* σ_{daily}, t = t_{daily} / 360.

Intuitively, if you look at the formula for

d1 = (log(S/X)+r*t)/(sigma*sqrt_t)+0.5*sigma*sqrt_t
d2 = d1 - sigma*sqrt_t)
c = S*cdf(d1)-X*exp(-r*t)*cdf(d2)

d1, d2 and c should not be effected by whether it is daily or yearly, since if you plug in the yearly parameters in the formula, there is simply no different from daily parameter, every parts that have changed all cancelled out. While still, I should test the result by program!

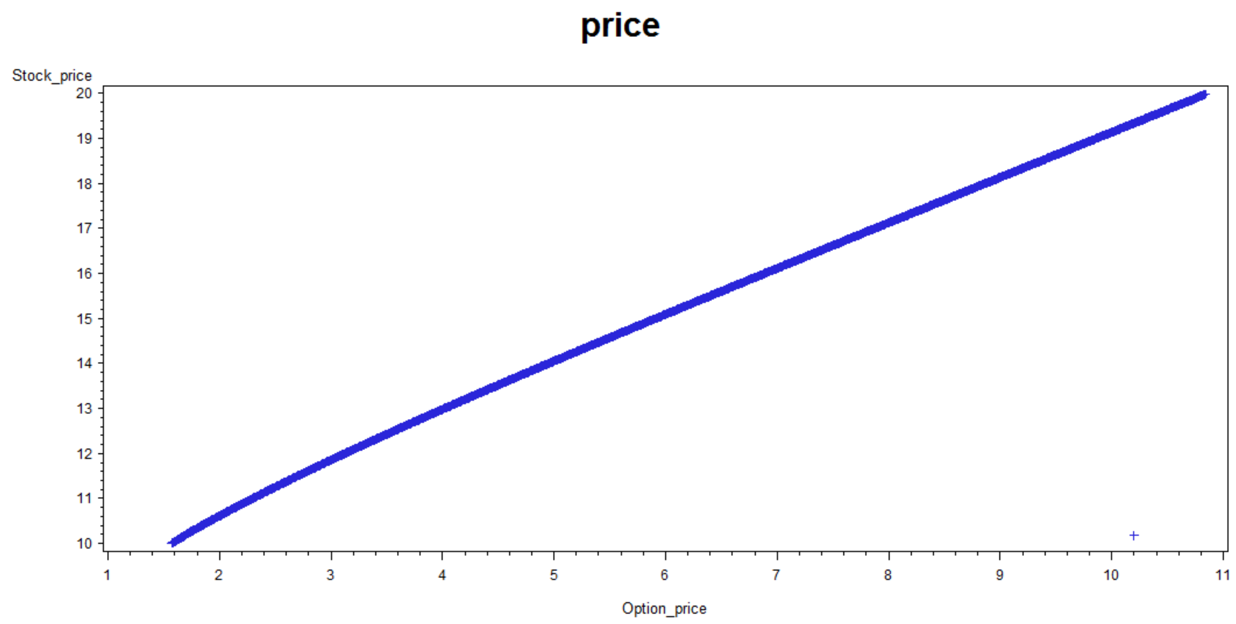
As the same reason, I just want to test the condition when t is relatively small.



```
>>> ================================ RESTART ============
>>>
>>> priceBS(10,5,0.001,0.03,.0000001)
Daily price!  5.0000000005
>>> ================================ RESTART ============
>>>
>>> priceBS(10,5,0.001*360,0.03*360**.5,.0000001/360)
yearly price!  5.0000000005
>>> |
```

From the result, we could see that there is no change on the result. So I assume there is no change on whether it is daily or yearly.

(3) Plot


price
Stock_price
20
19
18
17
16
15
14
13
12
11
10
1
2
3
4
5
6
7
8
9
10
11
Option_price

PART II

Problem 7

Solve analytically the following system of linear equations in (x, y, z, t) as function of (6):

$$\begin{aligned}
x + y + z + t &= 1 && (3)\\
2x + y + z + t &= 2 && (4)\\
x + 3y + z + t &= 3 && (5)\\
2x + y + 9z + 7t &= a && (6)
\end{aligned}$$

For which values of (6) this system has one solution? For which values of (6) it has no solution? For which values of (6) it has infinite solutions? For which values of (6), the problem is ill-conditioned? Explain your answers.

In order for function (6) to have one solution the matrix of (3),(4),(5) and (6) have to be nonsingular, if it is singular, (6) has no or infinite solutions.

Since det(A), A is the matrix comprise the system, is 64, for this system, no matter what is a, the system always has one solution, because it is nonsingular. It is always ill-conditioned since the condition number is always larger than 1.

Problem 8

What is the inverse of the following matrix?

$$\begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{a} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{b} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

How may this arise in computational practice?

The inverse of it, by doing the same as to convert it to Identity Matrix to Identity Matrix, we get the inverse matrix:

$$\begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -\boldsymbol{a} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & -\boldsymbol{b} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

Converting the matrix into its inverse is going to cause the 'zero' elements in the inverse matrix to change into junk numbers that are close to machine precision because in floating algorithm, sometimes a number is not what we perceive in math theory due to computational error or rounding error caused by floating computation.

Problem 9

Write an algorithm to efficiently invert any matrix A meeting the following conditions:

1. The matrix is non-singular
2. For each row, one and only one term is non-zero

You are not allowed to call the Gauss-Jordan algorithm. Explain why it works. Implement the algorithm in Python or C++. Discuss the speed of your algorithm compared with the Guass-Jordan one.

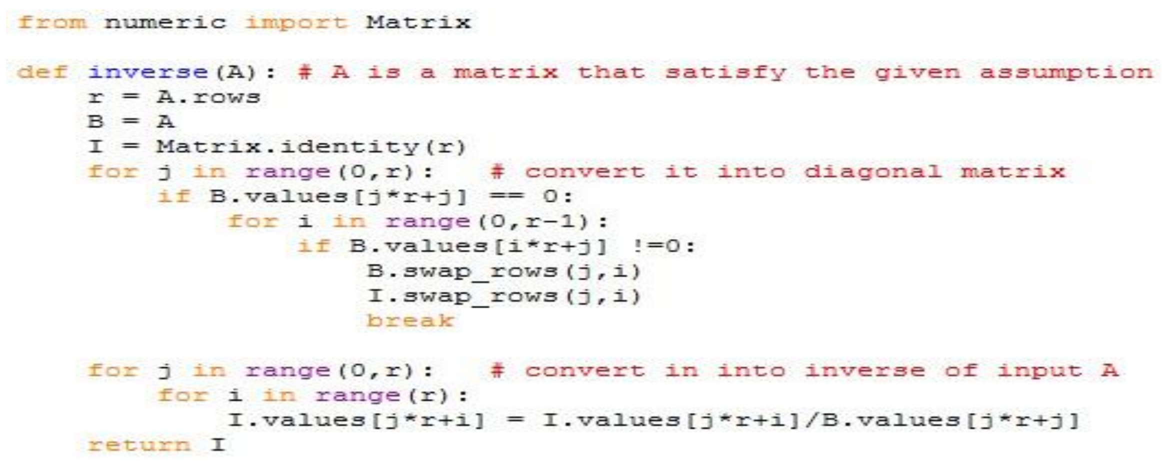
If the matrix is nonsingular, the matrix must (or after multiplied by certain permutation matrix) go like:

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & d \end{pmatrix} \text{(a, b, c, d are nonzero, respectively)}$$

It is impossible for any column to have more than one nonzero element, since in that case the matrix is singular.

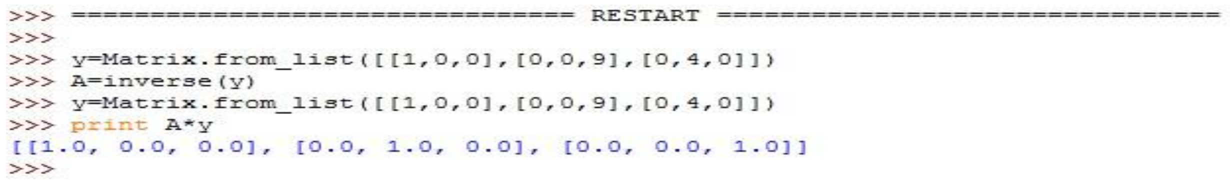
(1). Why it works (analyst of problem)
Ideally, we could do the same thing that converts a general matrix into an Identity Matrix to an Identity matrix to find the inverse, the Identity that has been changed is our inverse! So what I will do, I will create a Identity matrix that is the same size as the input matrix and do the same thing that convert the input matrix into identity(2) to the created Identity(1), then output Identity(1) as the inverse. To test the result, I will use numeric.Matrix class to test the sum of original and the inverse. That result should be identity!

(2). Code for problem
After detecting certain assumptions on that problem, I could do the following python script to solve for the solution:



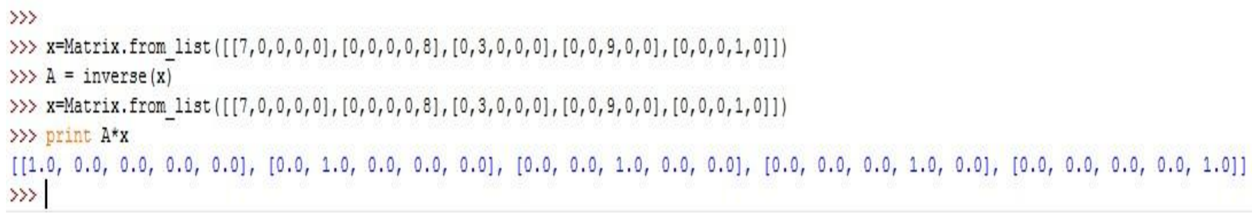
```
from numeric import Matrix

def inverse(A): # A is a matrix that satisfy the given assumption
    r = A.rows
    B = A
    I = Matrix.identity(r)
    for j in range(0,r):   # convert it into diagonal matrix
        if B.values[j*r+j] == 0:
            for i in range(0,r-1):
                if B.values[i*r+j] !=0:
                    B.swap_rows(j,i)
                    I.swap_rows(j,i)
                    break

    for j in range(0,r):   # convert in into inverse of input A
        for i in range(r):
            I.values[j*r+i] = I.values[j*r+i]/B.values[j*r+j]
    return I
```

For testing the result,



```
>>> ================================ RESTART ================================
>>>
>>> y=Matrix.from_list([[1,0,0],[0,0,9],[0,4,0]])
>>> A=inverse(y)
>>> y=Matrix.from_list([[1,0,0],[0,0,9],[0,4,0]])
>>> print A*y
[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
>>>
```

Another test also works!



```
>>>
>>> x=Matrix.from_list([[7,0,0,0,0],[0,0,0,0,8],[0,3,0,0,0],[0,0,9,0,0],[0,0,0,1,0]])
>>> A = inverse(x)
>>> x=Matrix.from_list([[7,0,0,0,0],[0,0,0,0,8],[0,3,0,0,0],[0,0,9,0,0],[0,0,0,1,0]])
>>> print A*x
[[1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0, 0.0], [0.0, 0.0, 0.0, 0.0, 1.0]]
>>> |
```

Problem 10

If A is an n*n matrix and x is an n-vector, which of the following computations require less work? Explain why.

$y = (XX^T)A$ (8)
$y = X(X^TA)$ (9)

$y = (XX^T)A$ requires n**2 times computation of multiply, n*n matrix multiplied by n*n matrix, which requires n*n*n multiplication +n*n*(n-1) addition. 2* n**3

The second formula requires n*n multiplication +n*(n-1) addition, then needs n**2 multiply. The total is 3*n**2-n

The second formula is better.

Problem 11

For a square matrix, what is it easier to compute the 1-norm or the 2-norm? Why?

1-norm is easier to compute than 2-norm, since for 1-norm, you just have to sum up every column of the matrix A and find out which is the largest, that is the value of 1-norm. However, 2-norm is the the square root of the largest eigen value of the matrix A^TA, which is harder to get.

Problem 12

Suppose A is a positive definite matrix. Show that A is non-singular. Show that the inverse of A is also positive definite.

For a positive definite matrix, it satisfies that $X^T A X > 0$ for all X != 0. However, if A is singular, then there must be that Az = 0, z is any vector, which means that $X^T A = 0$ or XA = 0, thus it does not satisfy $X^T A X > 0$. Therefore, A cannot be singular.

$A^{-1} = I / A$. Since A is nonsingular, I /A is also nonsingular. Consequently, A^{-1} is also nonsingular.