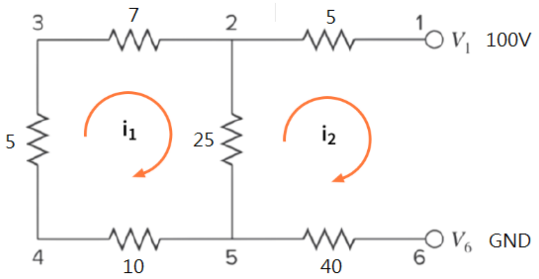
HW2 B06301054 林祐安

There are totally 8 files in the compressed directory, NA_HW2_Q1, NA_HW2_Q2, NA_HW2_Q3, epsilon.mat, epsilon1.mat, epsilon2.mat, noise1.mat, and noise2.mat. If any one is missing or damaged, please download it here.

1. Run NA_HW2_Q1.m

(a)



$5i_1 + 7i_1 + 25(i_1 - i_2) + 10i_1 = 0$

$25(i_2 - i_1) + 5i_2 + 40i_2 = 100$

Or

$47i_1 - 25i_2 = 0$

$-25i_1 + 70i_2 = 100$ ---(*)

Substitute $u, v\ for\ i_1, i_2$ respectively to avoid confusing sequence with variable name.

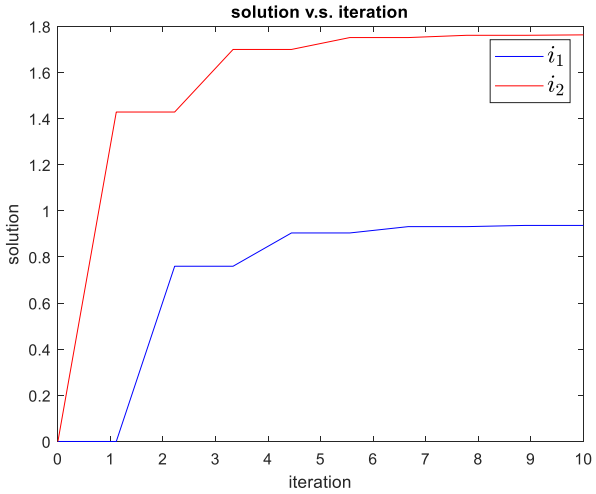
$$47u - 25v = 0 => u_{i+1} = \frac{25v_i}{47}$$

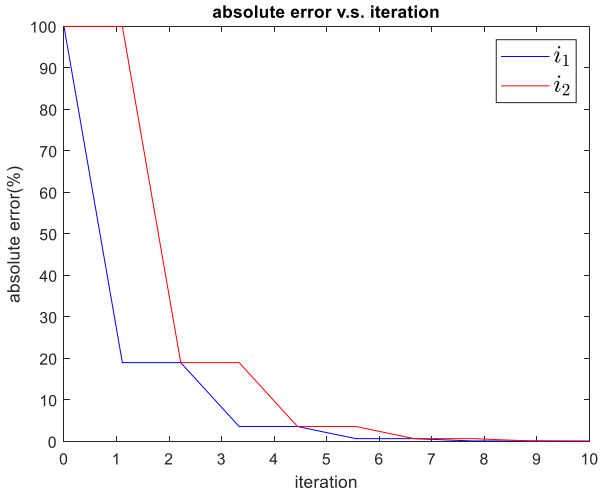
$$-25u + 70v = 100 => v_{i+1} = \frac{100 + 25u_i}{70}$$

We calculate the solutions solved from the above equation (*), $u = 0.9381, v = 1.7636$, for the sake of absolute errors.

Let's start doing Jacobi method. Initial guess $\begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

After running this file, *NA_HW2_Q1.m*, the first figure describes the recursive process.





The figures indicate that both of u and v converge, and furthermore they roughly approach to $0.936865\ and\ 1.76317$ respectively.

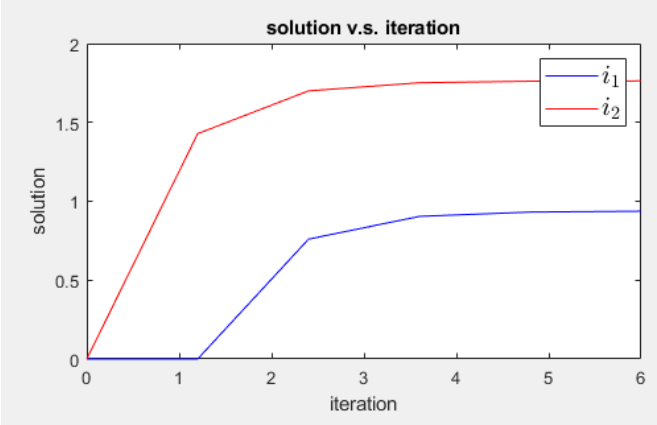
(b)

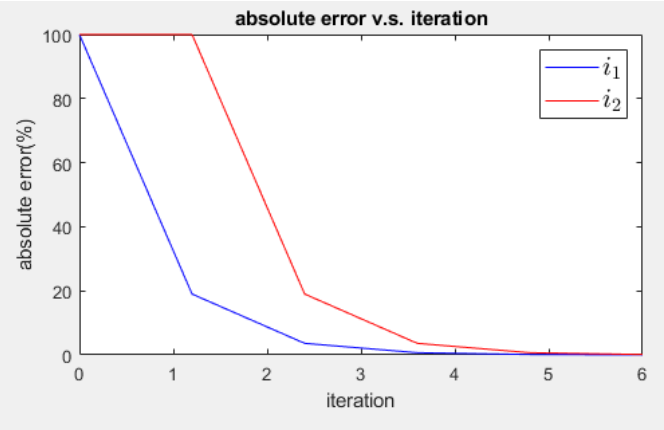
Let's revise somewhere in previous problem to adapt Gauss-Seidel method.

$$u_{i+1} = \frac{25v_i}{47}$$

$$v_{i+1} = \frac{100 + 25u_{i+1}}{70}$$

The second figure after running the file describes the recursive process using Gauss-Seidal method.





The final values are same as ones derived from Jacobi method. However, the rate of convergence of Gauss-Seidel method is higher than that of Jacobi method.

(c) Reference (1:02:00 start)

Jacobi method with relaxation $\lambda = [0.3,0.7,1.2]$

$$u_{i+1} = \lambda\left(\frac{25v_i}{47}\right) + (1-\lambda)u_i$$

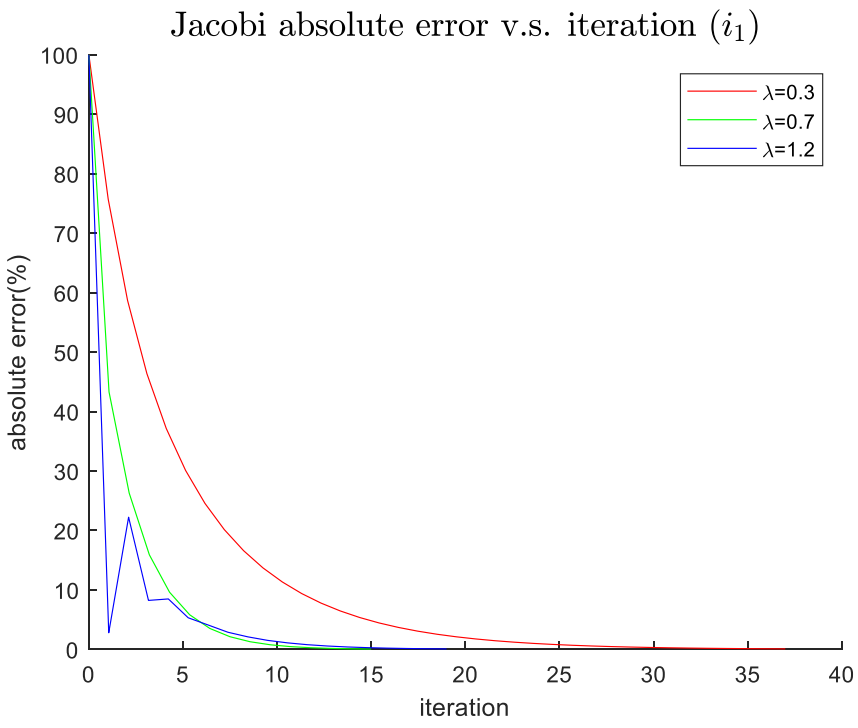
$$v_{i+1} = \lambda\left(\frac{100 + 25u_i}{70}\right) + (1-\lambda)v_i$$

Gauss-Seidel method with relaxation $\lambda = [0.3,0.7,1.2]$

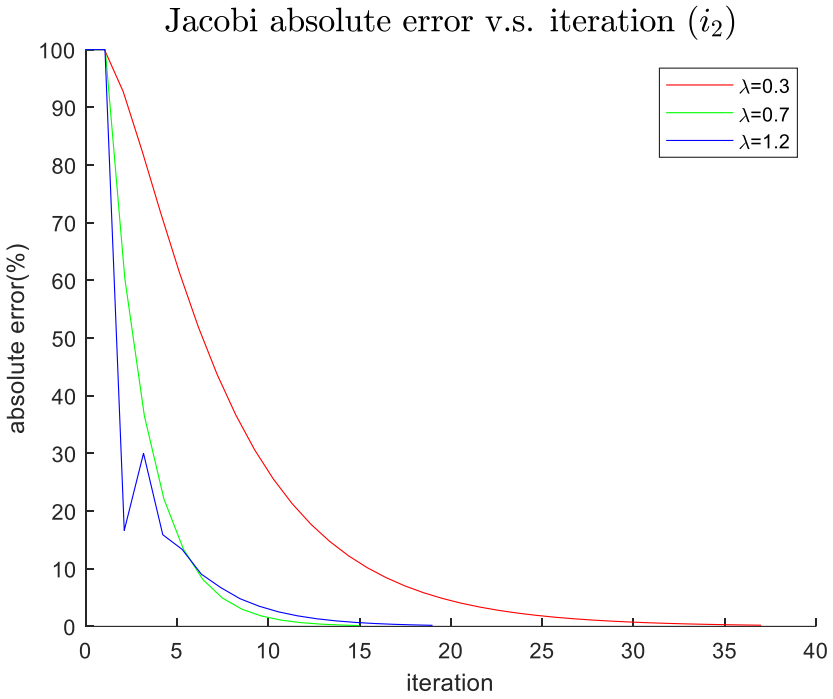
$$u_{i+1} = \lambda\left(\frac{25v_i}{47}\right) + (1-\lambda)u_i$$

$$v_{i+1} = \lambda\left(\frac{100 + 25u_{i+1}}{70}\right) + (1-\lambda)v_i$$

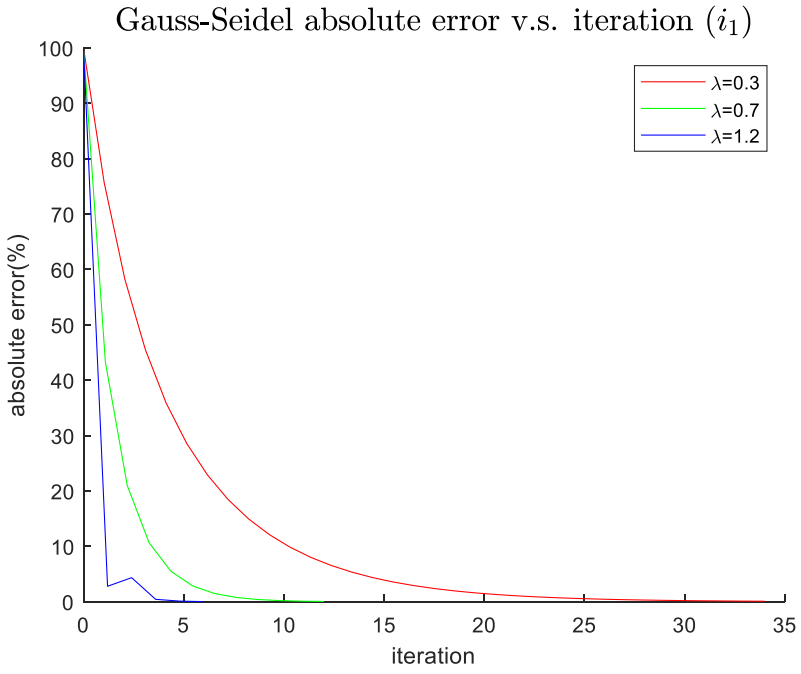
Last four figures represent the difference processes between distinct λ and methods .



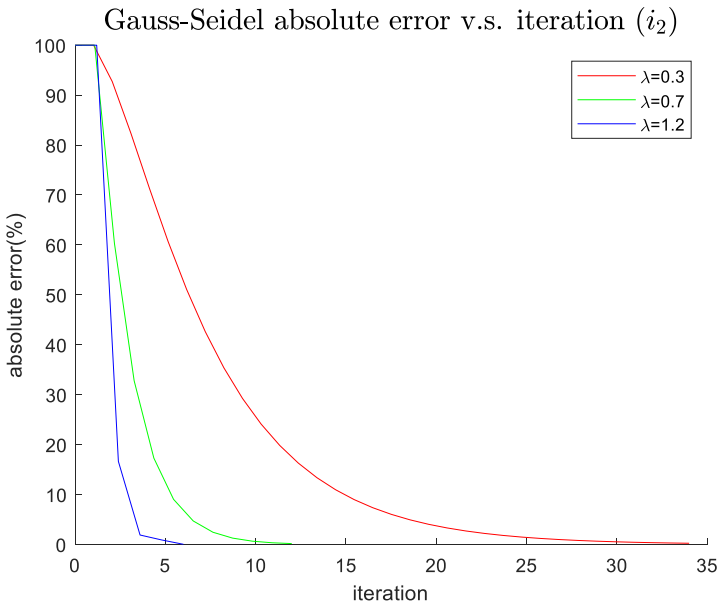
(fig.4 Jacobi method i_1 with different λ)



(fig.5 Jacobi method i_2 with different λ)



(fig.6 Gauss-Seidel method i_1 with different λ)



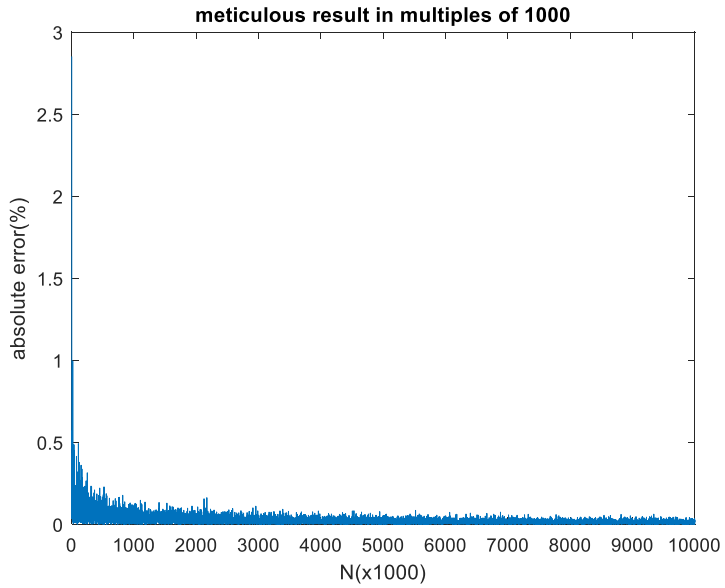
(fig.7 Gauss-Seidel method i_2 with different λ)

Base on these plots, we conclude that the rate of convergence is proportional to bounded λ, whereas if λ is even larger then the system breaks down. On the other hand, the rate is slow at first, but is accelerated with appropriate increase of λ.

2. Run NA_HW2_Q2. There are three pre-trained files named '*epsilon.mat*' , '*epsilon1.mat*' and '*epsilon2.mat*', which will be introduced afterwards. You can alternatively comment out it and additionally run functions 'meticulousrun()' , 'varrun()' and 'eachrun()'. Warning: the first two function will take you about 15 mins depending on your device.

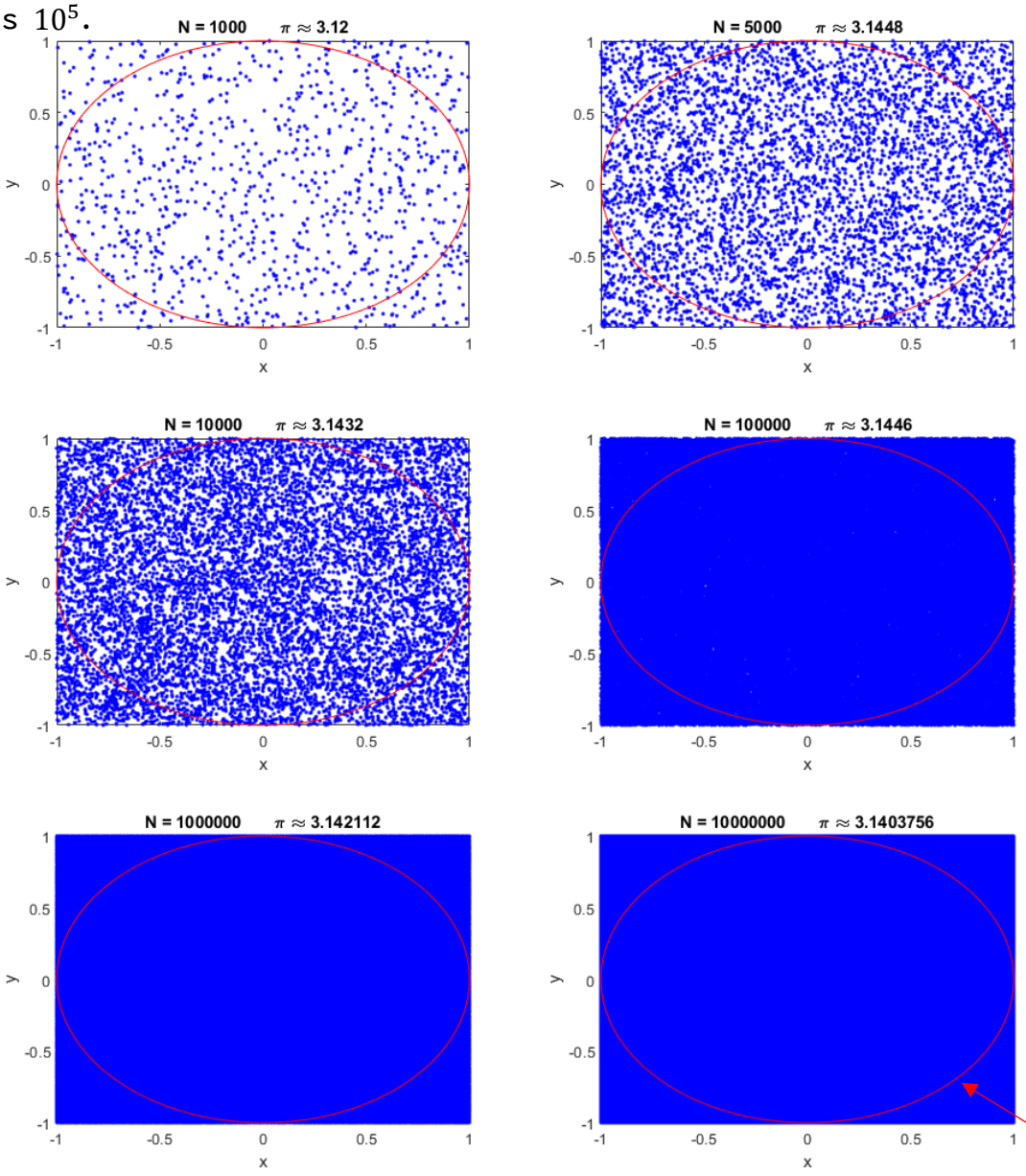
Let's decide the range of N(our 'beans') initially, N=[1000,10^7], then estimate π value when N is a multiple of 1000,i.e. N=linspace(1000, 10^7,1000). An 1 by 10^4 vector, '*epsilon*', is built to store absolute errors. Function: 'meticulousrun()'

Note that execution time is an exponential growth. When size of '*epsilon*' is growth, $R^{1\times100}, R^{1\times1000}$ *or* $R^{1\times10000}$, the loop takes 8.01s, 26.87s, 471.96s to be finished. Even though I had used 'parfor ' loop in order to accelerate computation (1176s without parellel computing for the largest case). Hence, I no longer increase N to avoid heavy workload. To be convenience, absolute errors has been saved as *epsilon.mat*.



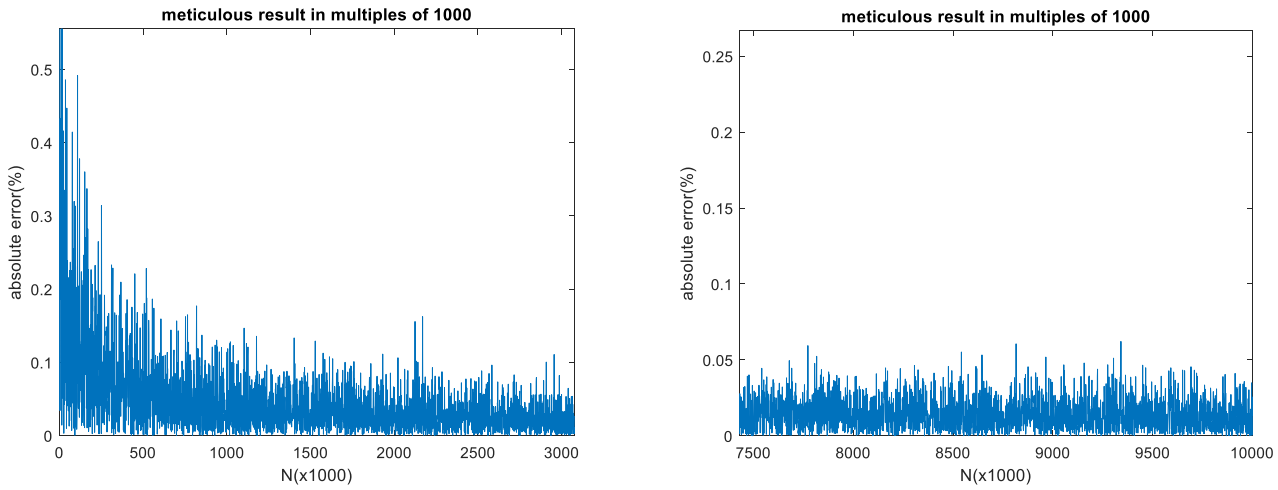
(fig. a)

In fact, there are many ways to reduce running time. For example, you can increase the increment from 1000 to 10000 at the expense of precision once N exceeds 10^5.



Round !

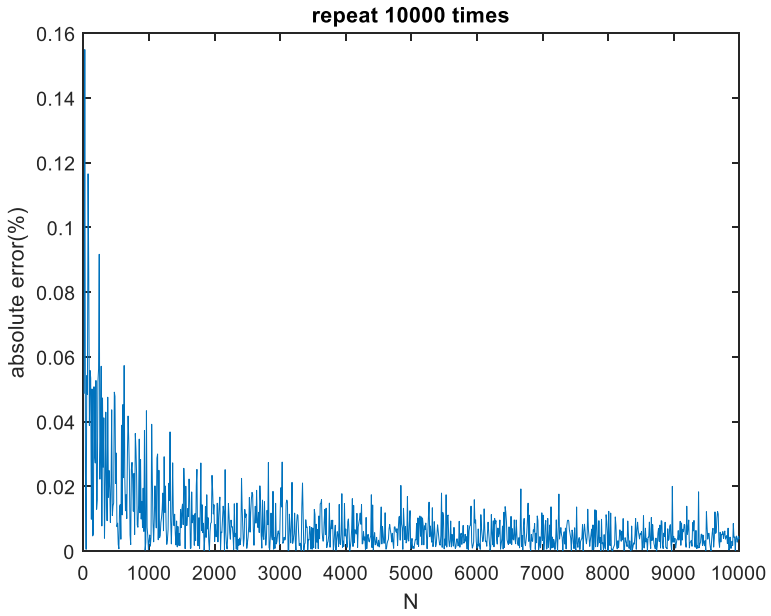
(using different N to approximate π value)



(zoomed fig.)

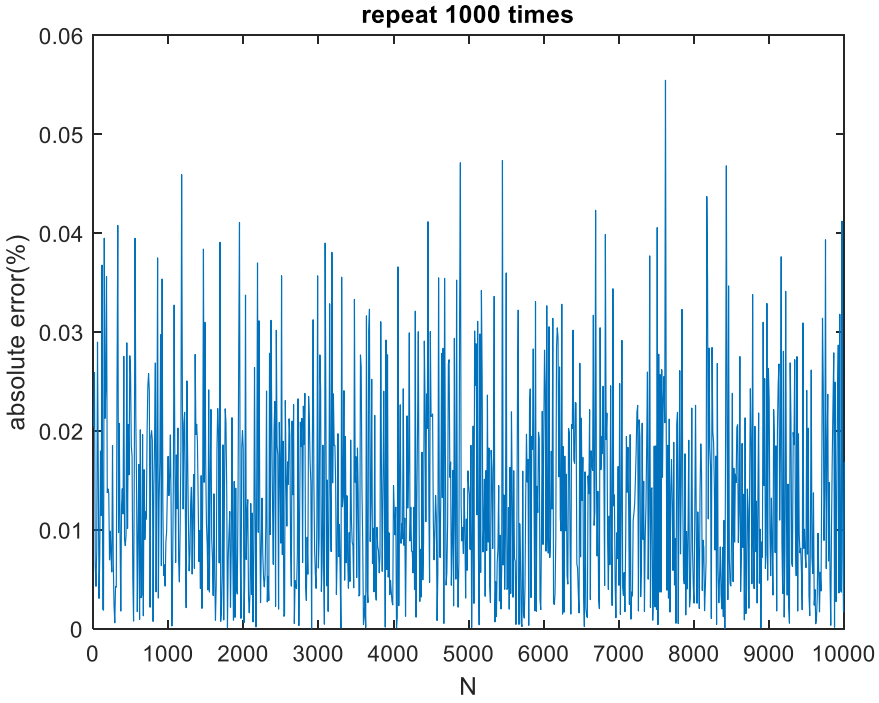
The curve seen in the zoomed figure is oscillating dramatically at first. If you have super good luck, all points may be distributed in the unit circle. Get $\pi = 4$, or even equaling 0 unfortunately. Result is entirely a matter of chance when N isn't large enough. According to the law of large numbers, the errors must converge and be smaller as N increases. Obviously, our finding complies with the theorem.

For finding a convergence function, we only discuss the fewer samples(N) case afterwards. Let's only consider N=linspace(10,10000,1000). Note linspace(Start,Terminal, # sample points). Repeate doing Monte Carlo method 10000 times on each sample point and calculate the average of absolute errors.



(fig. b)

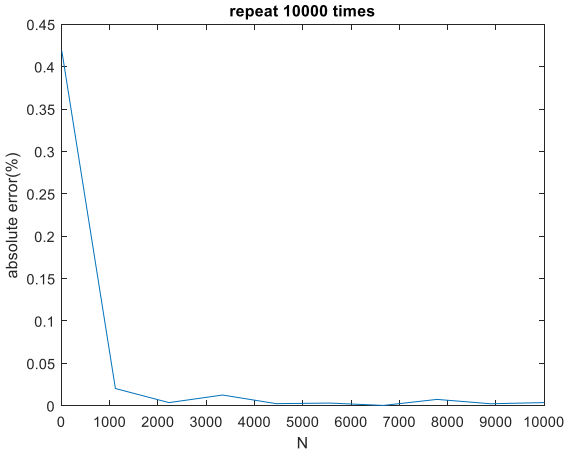
It's still oscillating. Improve it such that the product of repeated time and N is fixed, which means trying more times as N is small. Assume the product, *productnum*, is 10^7. Will the curve be more smooth? Function: 'varrun()'



(fig. c)

Well…it costs a lot of time, so I have saved errors as *epsilon1.mat*. Even worse? It seems terrible but in fact the outcome follows the law of large numbers due to several attemps. Extremely small errors bring on oscillating graph.

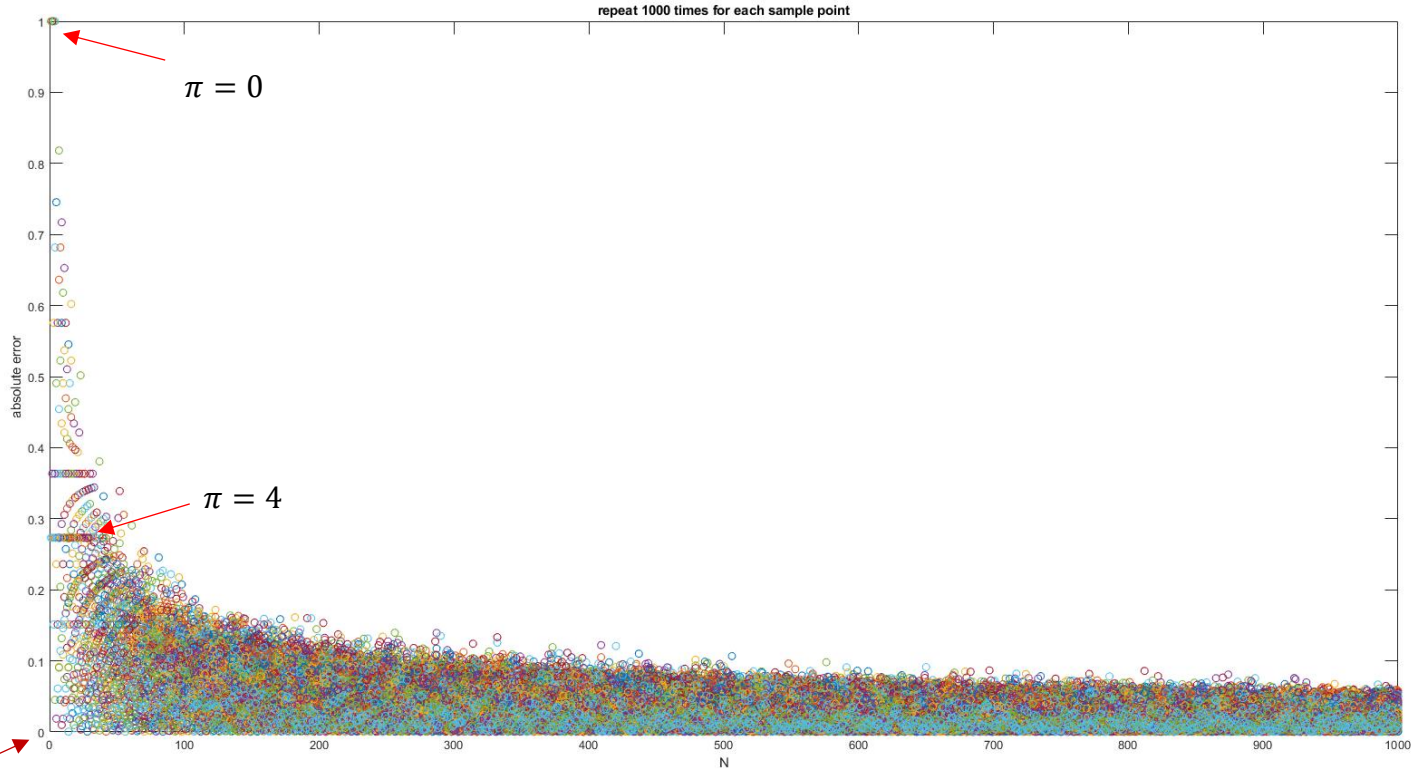
If we intend to obtain a smooth curve, reducing the sampling rate may be a better way. Try another idea , where N=linspace(10,10000,10).



(fig. d)

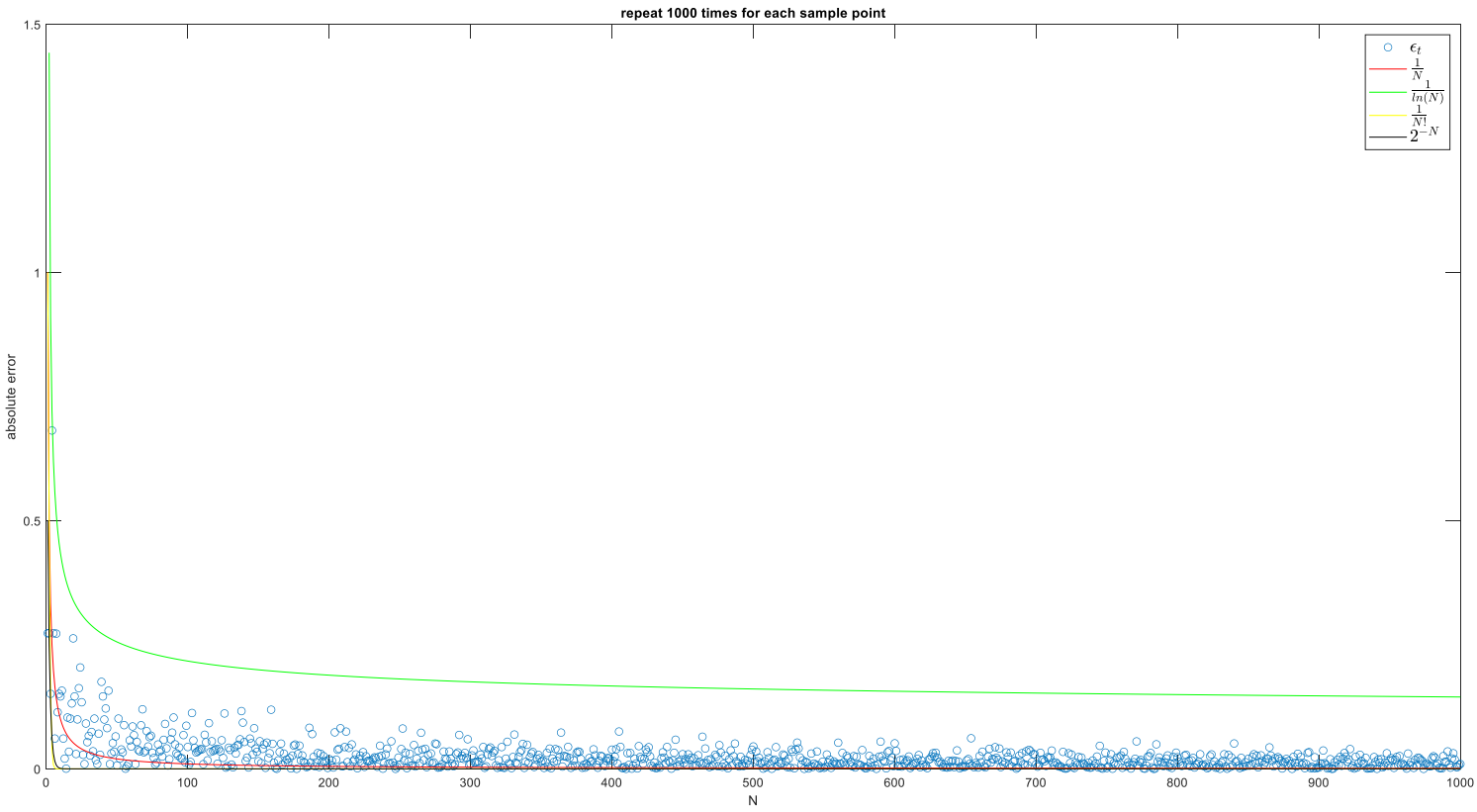
In this case, sacrificing values in the range of (10,1100) is an unfavorable situation that we unable to fit a function in the beginning. The fact of the matter is the error function goes down much more quickly at the outset.

It's time to show the final case. After some trial and error, to achieve a balance between precision and time consuming, repetition time and appropriate sampling rate are matters. Let N=linspace(1,10000,100) and repeated time=1000. Starting at 1 is for the purpose of precision. To avoid following the law of large numbers, repeated time requires to be fewer. We alternatively plot the procedures on each sample point for each repeated epoch. Errors have been saved as *epsilon2.mat*.Function:'eachrun()'



$\pi \approx 3.14$

(fig. e repeat spreading points 1000 times and print it out for each epoch)



(fig. f)

The above figure discribes the error curve compares with different kinds of convergence. We can conclude the order of convergence may be $O(\frac{1}{N})$. In mathematics, by the central limit thorem, Monte Carlo method displays $\frac{1}{\sqrt{N}}$ convergence. [wikipedia]

In sum, 首先我們必須分成兩個部分討論：每個 **N** 只做一次、每個 **N** 重複做多次

1. 根據大數法則，**N** 愈大，則趨勢收斂，**fig. a** 可以佐證。當 **N** 小時吃運氣，容易造成極端值，若 **N** 取樣間隔太近，就會看起來震盪劇烈，撒的點若夠多，即 **N** 大時，運氣問題就能減緩。

2. 根據大數法則，就算 **N** 很小，在固定 **N** 的情況下，只要我們嘗試多次，其結果也會收斂，因此我們可以從 **fig. c** 看到，就算只有撒 **10** 個點，做了一百萬次後其結果值也約等於 **3.14**。

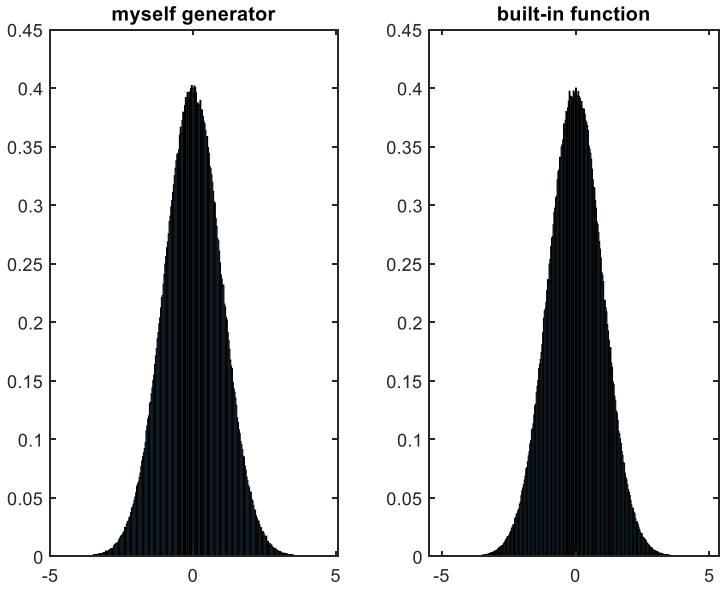
3. 綜合以上兩點，若我們找尋收斂函數時所使用的誤差值是經過平均計算的，也就是將重複做的取平均值，獲得一條曲線，如 **fig. b**，會因為重複做多少次而影響找到的收斂函數，重複做愈多次會使得誤差值愈小而愈平緩，愈少次則愈極端愈容易找到一條收斂曲線；而如 **fig. d**，由於誤差收斂速度非常快，間隔過大使資訊遺失過多，影響我們找到的是何種收斂函數，也因此 **fig. e** 才調整為重複做 **1000** 次並採取適當區間的點分布圖方便我們找尋收斂函數。

4. 從 **fig. f** 看到最接近的函數應是$O(\frac{1}{N})$，當然，我們很難透過 **fig. f** 決定是$O(\frac{1}{N})$、$O(\frac{1}{N^2})$或$O(\frac{1}{\sqrt{N}})$，這部分需依靠數學分析，無法從圖斷定。

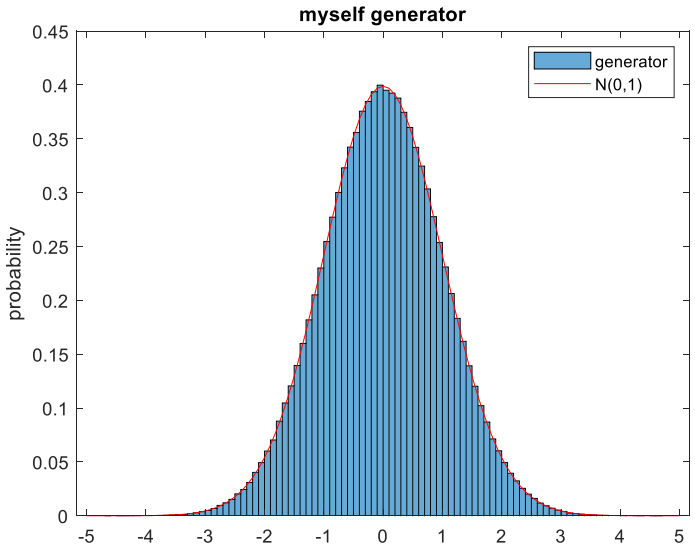
3. Run NA_HW3_Q3. There are two files, *noise1.mat* and *noise2.mat*, to guarantee your result is same as mine. Of course, you can comment out them and try to observe differenct results by yourself.

(a)

Box-Muller transform is performed instead of reject method to generate a ramdom normal distribution with mean at 0 and standard deviation of 1. There are one million points. Note: built-in function is randn(1,N)

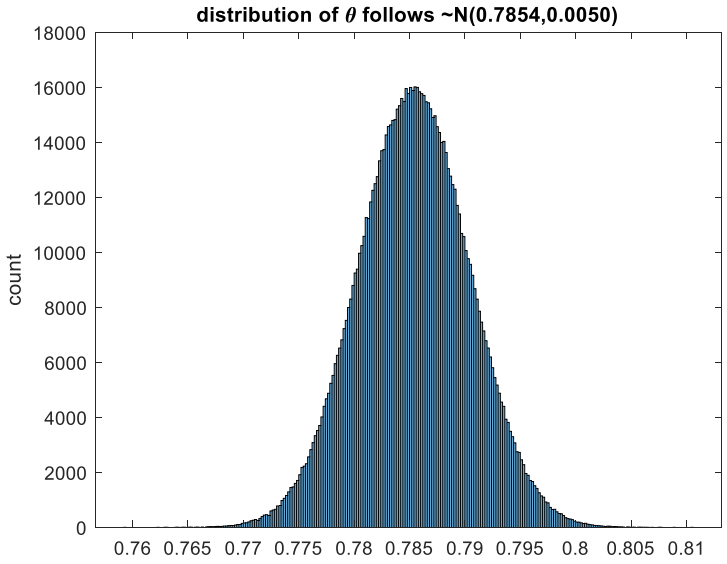


Verify our RNG in the range of [-5,5] with the pdf function $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$.

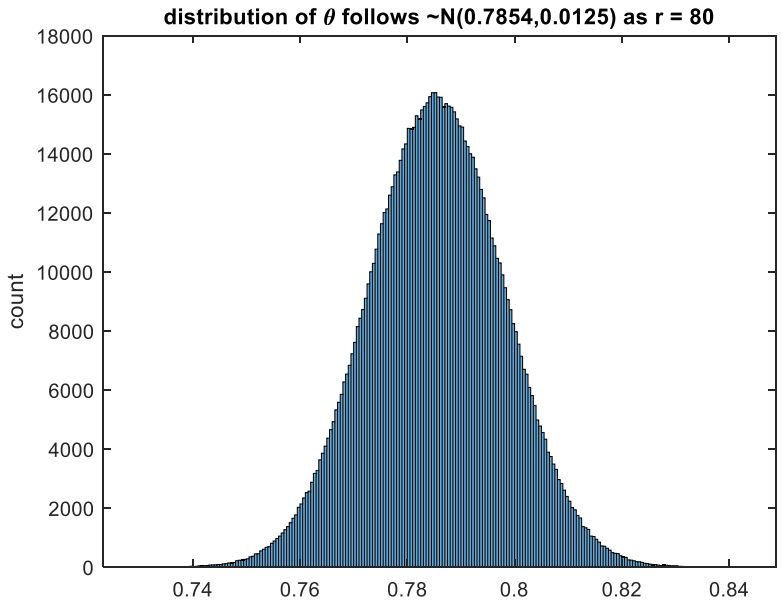


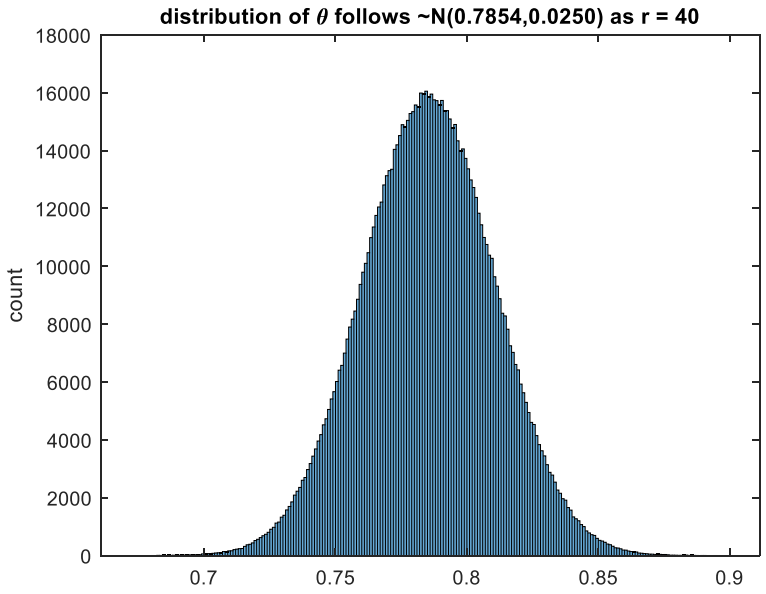
(b)

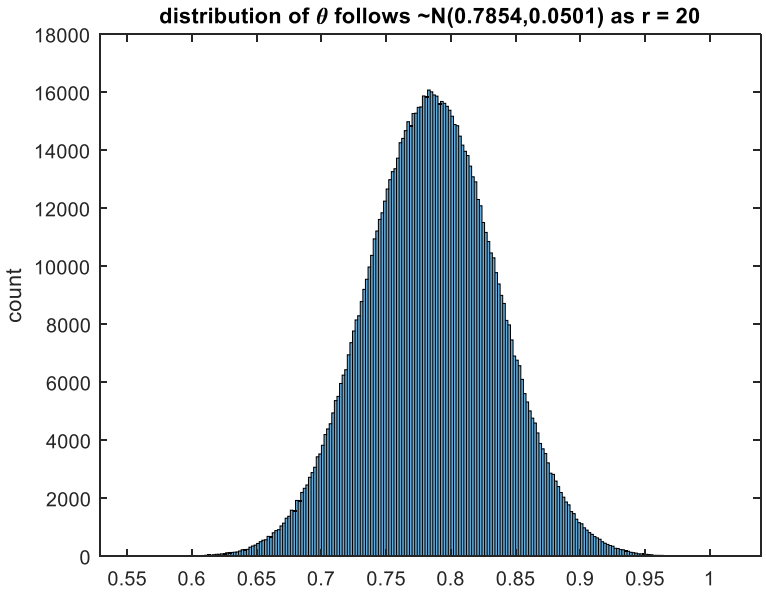
The distribution of θ follows the normal distribution $\sim N(\mu, \sigma) = \sim N(0.7854, 0.005)$.

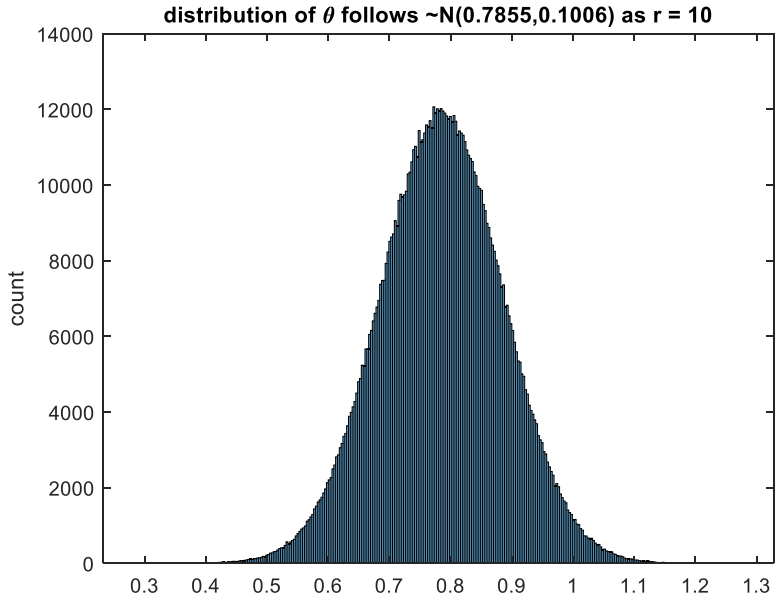


(c)

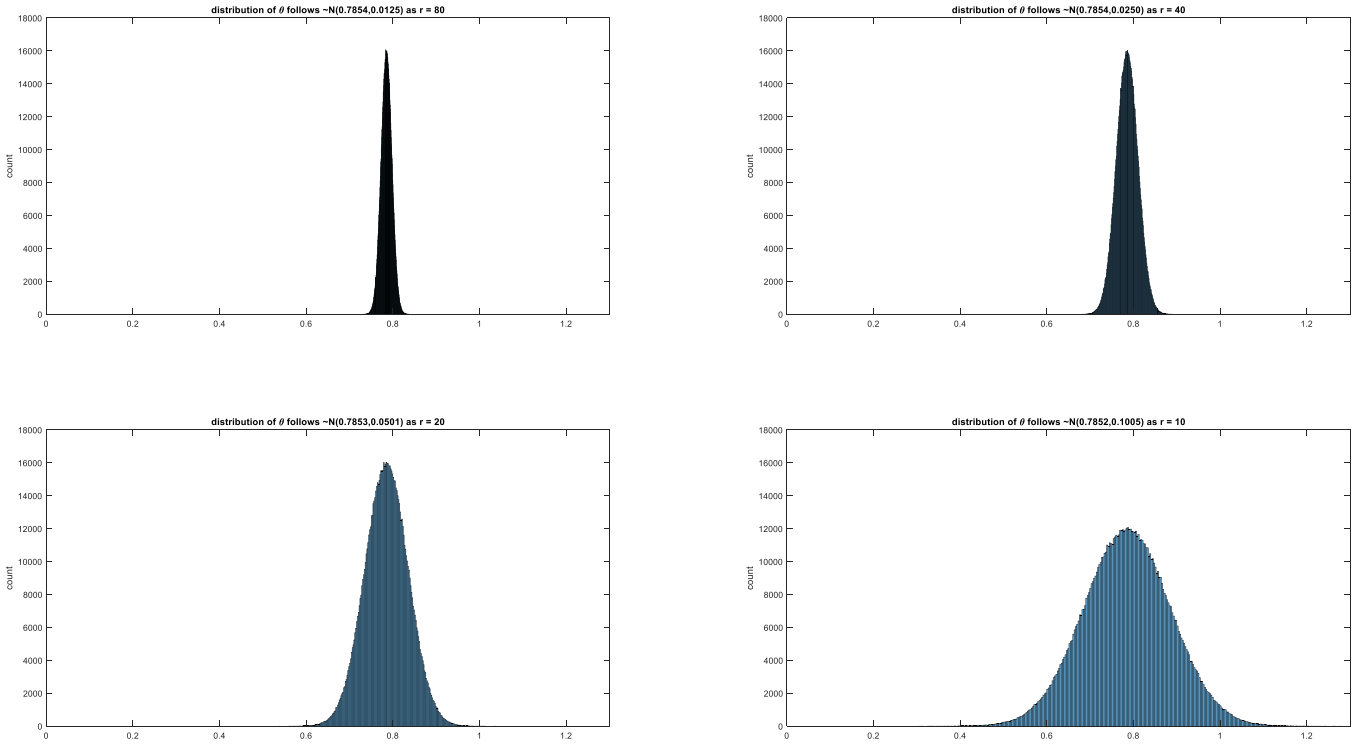



distribution of θ follows ~N(0.7854,0.0250) as r = 40
count
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0.7
0.75
0.8
0.85
0.9


distribution of θ follows ~N(0.7854,0.0501) as r = 20
count
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0.55
0.6
0.65
0.7
0.75
0.8
0.85
0.9
0.95
1


distribution of θ follows ~N(0.7855,0.1006) as r = 10
count
14000
12000
10000
8000
6000
4000
2000
0
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3

According to the above figures, the shorter vector the larger standard deviation. It's intuitive that the shorter verctor is more easily affected by noise.



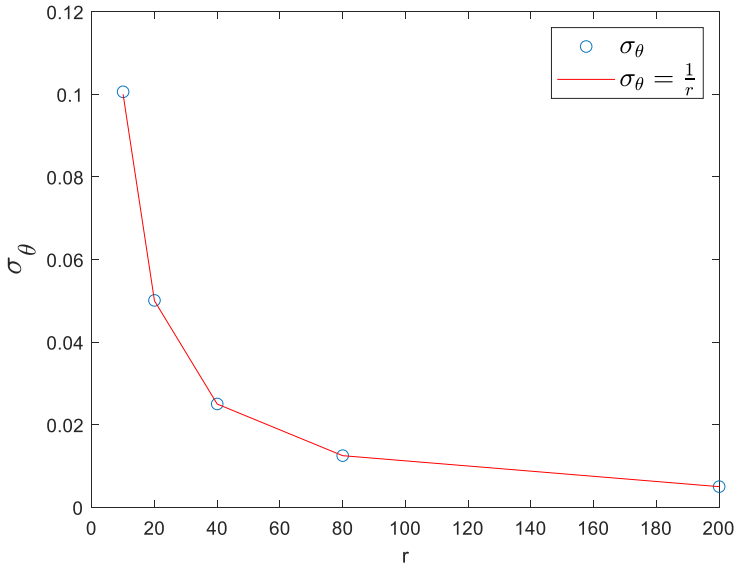
(in the same scale)

(d)

In the previous problem (c), we know that the shorter vector the larger standard deviation, which implies that they may be complement. Moreover, the following table indicates the product of length and standard deviation is a constant.

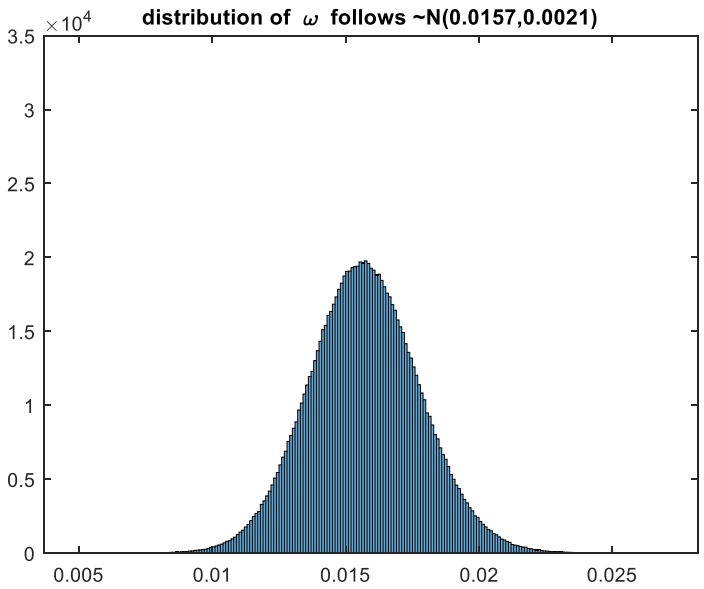
$$r \times \sigma_\theta = 1 => \sigma_\theta = \frac{1}{r}\ ,where\ r\ is\ the\ length\ of\ vector.$$

r	10	20	40	80	200
σ_θ	0.1006	0.0501	0.025	0.0125	0.005
$r \times \sigma_\theta$	1.006	1.002	1	1	1

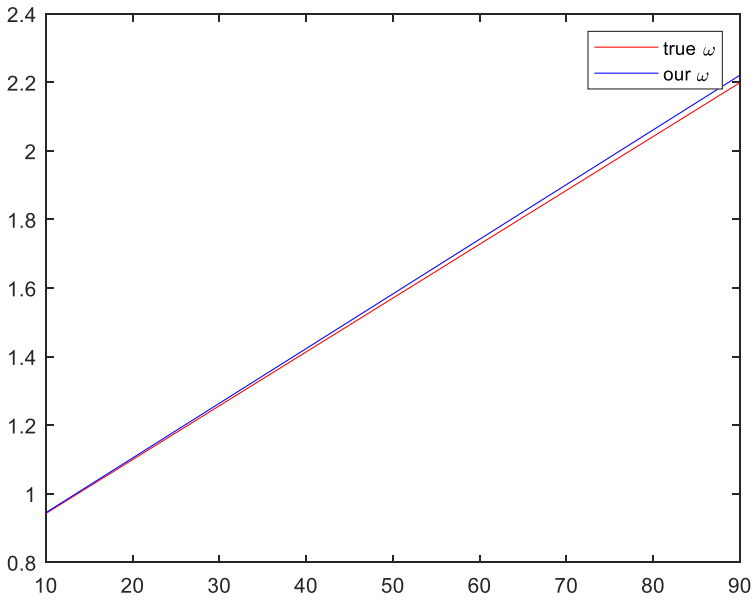


(e)

ω *follows the normal distribution of mean* $= 0.0157$ *and* $\sigma = 0.0021$.



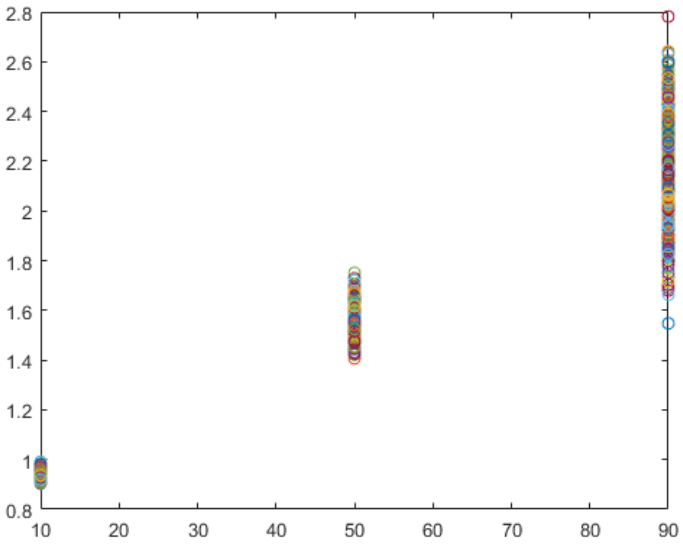
As the problem discribes $\omega = \frac{\pi}{200} \approx 0.0157$. Fortunately, the mean of the distribution of ω is 0.0157. For the convenience, we can conclude the slope of regression function, $\theta = \omega t + \theta_0$, is 0.0157, although the mean of a distribution may be NOT the unique peak. However, in this code file, I set where the max count exists as our finding slope ω.



If we increase decimal precision , or plot the true ω versus another ω determined by regression, there may be a little skewed. The two lines are sometimes very close or even overlap without zoomed because N is very large. If you set N=1e4, they skew more obviously. Before next problem, we observe the scatter of θ in advance to help us design a weighting matrix.

Please do NOT run the code when N=1e6, plot function probably breaks down!

```
% Do NOT run the following code! Matlab will break down!
% if you want to check it, please reduce N at line five,
% such as N=1e4,then rerun the kernel.
%%% figure
%%% plot(t(1),S_theta(1,:),'o',t(2),S_theta(2,:),'o',t(3),S_theta(3,:),'o')
```

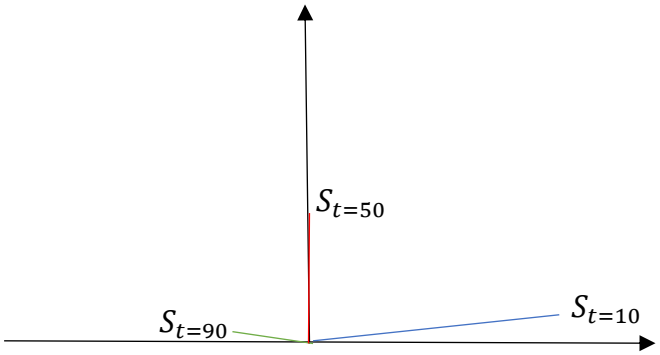


(fig. g t-θ scatter in a different condition,N=1e4)

(f)

The purpose of weighting function is to equalize noise. As seen in the figure, noise affects our signal significantly at t=90. Why?

In problem(d), we know the noise effects are inversely proportional to the length of signals.

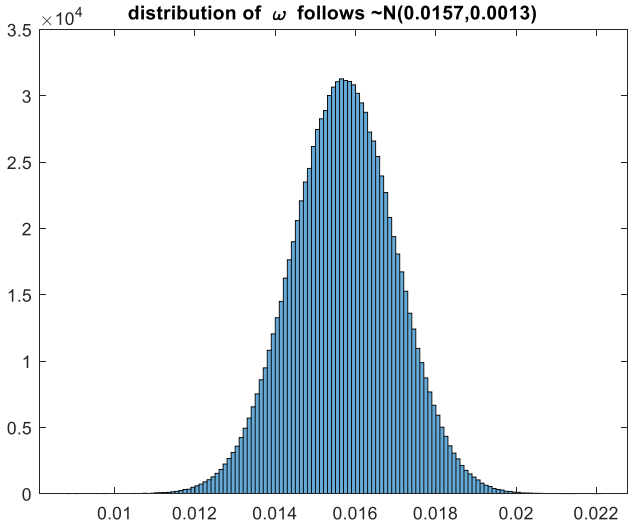


According to $S(t) = S_0 e^{-\frac{t}{T_2}+i(\omega t+\theta_0)}$, the longer time gives rise to the shorter length. Therefore, we can explain what the fig. g describes. In order to reduce the noise effects, let's shrink scale at t=90 and 50 to gather each group, but more significantly at t=90. The weighting matrix is $W = \begin{bmatrix} 0.6 & 0 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$, then $WY = WA\hat{X} \Rightarrow Y' = A'\hat{X} \Rightarrow$

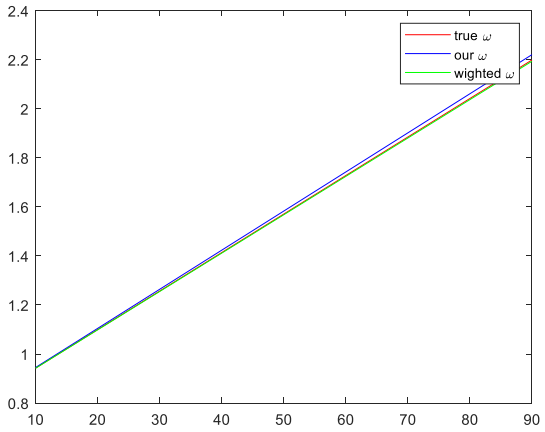
$\hat{X} = (A'A)^{-1}A'Y$, where $Y\ is\ a\ matrix\ M_\theta \in R^{3\times N}$, $A = \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ 1 & t_3 \end{bmatrix} = \begin{bmatrix} 1 & 10 \\ 1 & 50 \\ 1 & 90 \end{bmatrix}$, $X = \begin{bmatrix} \theta_0 \\ \omega \end{bmatrix}$, and $\hat{X}$ is

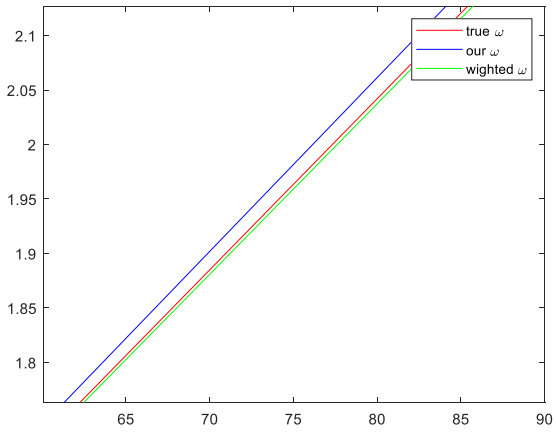
a least sqaure solution.

After implementing the weighting function, the distribution of ω becomes narrower and taller. It implies that the regression model is more comprehensive.



Let's finally investigate the difference of ω between with and without a weighting function.

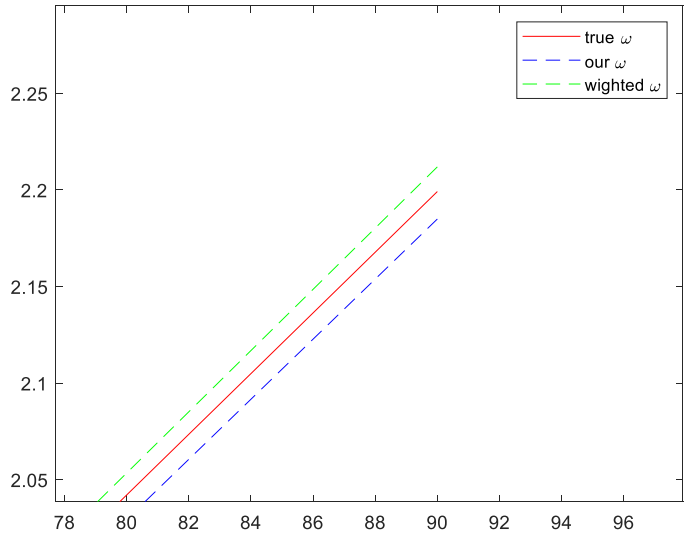




(original size) (zoom in the top-right side)

The outcome is worthy of our expectation. By the way, if we design this weighting matrix in an opposite direction, such as $W = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 0.6 \end{bmatrix}$, the distribution becomes shorter and more flattened.

In additionally, the weighting matrix should be designed on a case-by-case; otherwise, the weighting regression model does NOT have a better performace or even worse.



(another cases without loading noise files then rerunning it)

Because we get a different distribution of data for each experiment, they are generated randomly.