COMPUTATIONAL LAB

LAB 1

INTRODUCTION

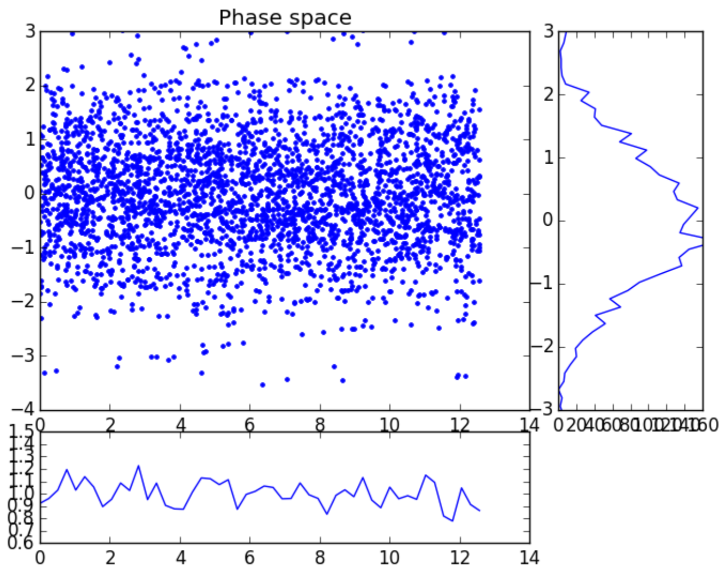
WHAT LANDAU DAMPING IS.

Download e epcld.py

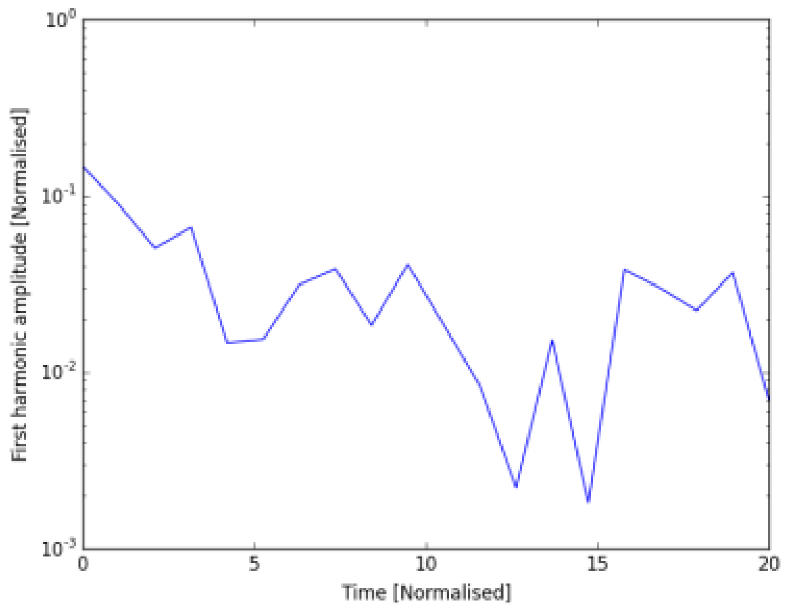
Opened cd documents etc

Opened terminal and run >> % epcld.py

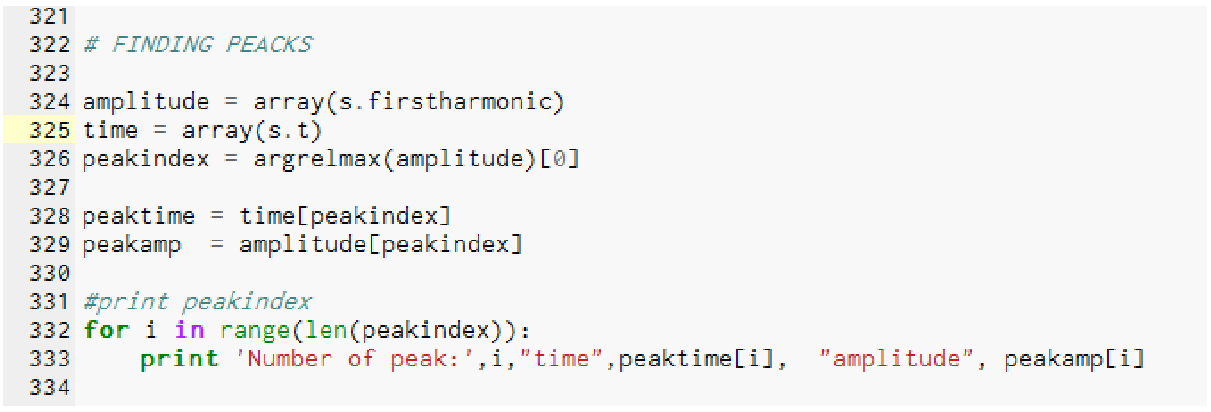
This produced animated in phase space then plot first harmonic amplitude.



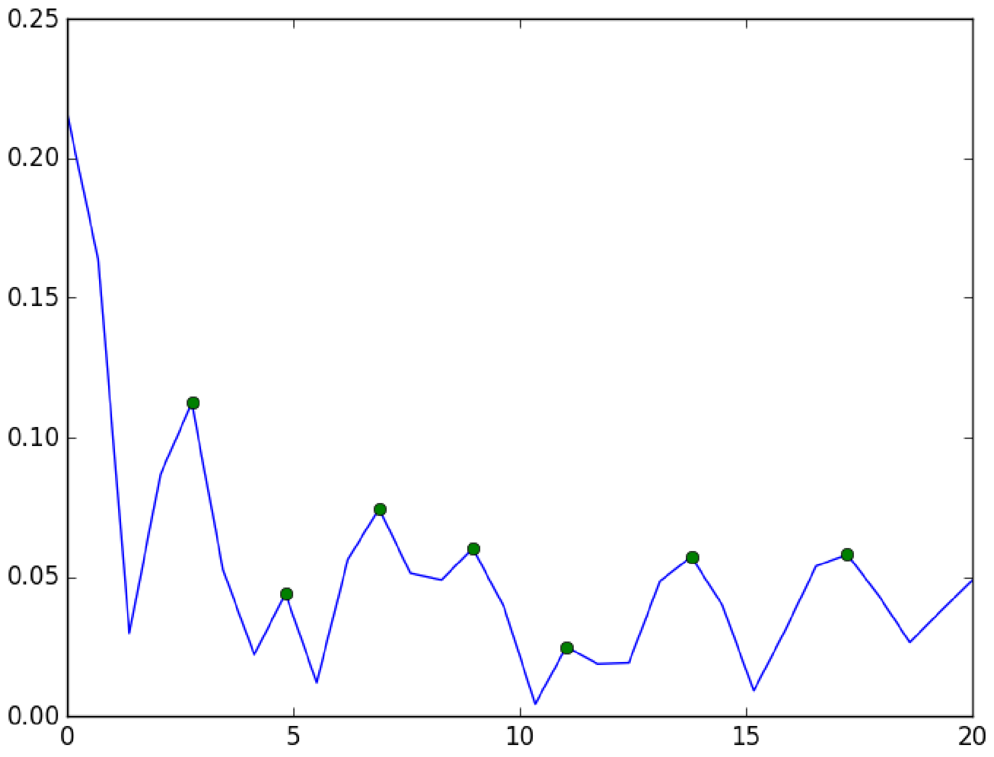
This graphs shows moving particles.



This plot shows amplitude vs time in normalised units
The wave decades over time until it reach a noise level.
To analyse this function of amplitude of first harmonic, we should find peaks of the line.
For this we alter the code.



```

# FINDING PEACKS

amplitude = array(s.firstharmonic)
time = array(s.t)
peakindex = argrelmax(amplitude)[0]

peaktime = time[peakindex]
peakamp  = amplitude[peakindex]

#print peakindex
for i in range(len(peakindex)):
    print 'Number of peak:',i,"time",peaktime[i],  "amplitude", peakamp[i]

```



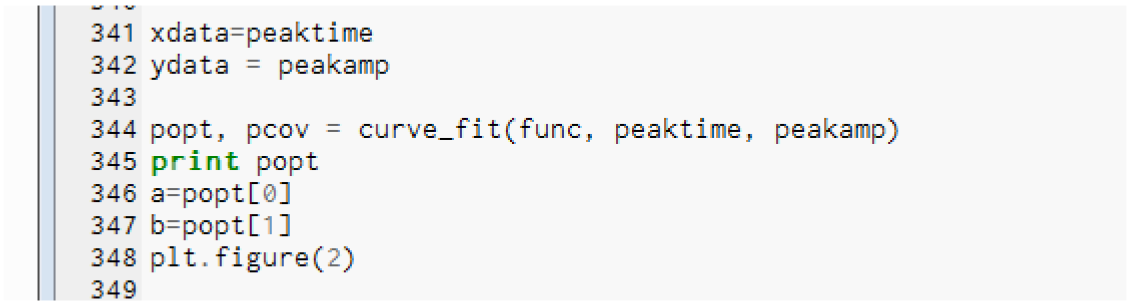
This figure shows amplitude of first harmonics.

What we did is found peaks of the line and show the peaks by green dots.

We used “argrelmax” python tools.

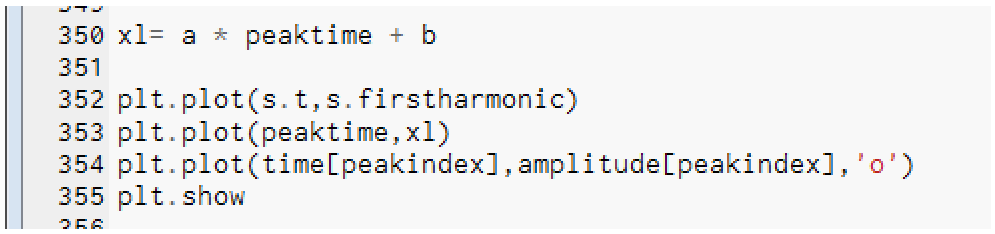
Now we need to find a line which extrapolate these dots.

Here is the code.



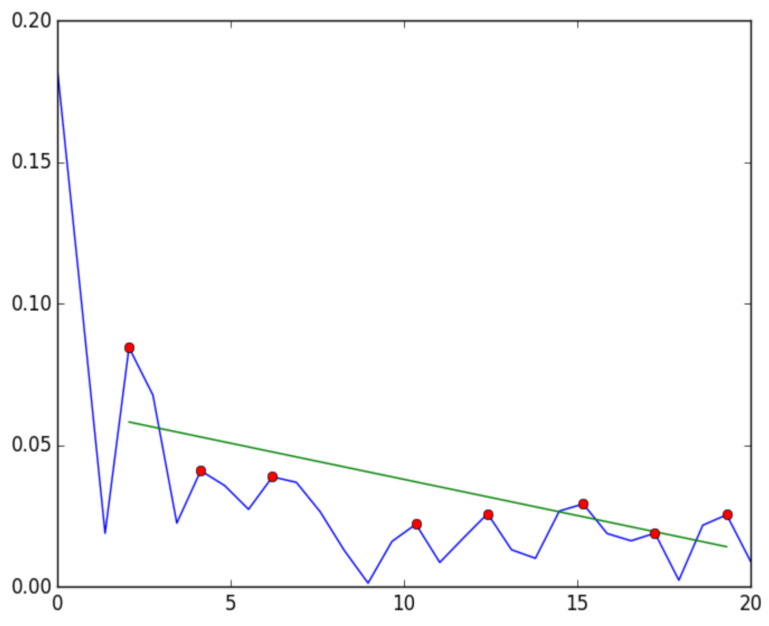
```
xdata=peaktime
ydata = peakamp

popt, pcov = curve_fit(func, peaktime, peakamp)
print popt
a=popt[0]
b=popt[1]
plt.figure(2)

```

The code finds the coefficients for a line which is the best fitted for the dots.



```
xl= a * peaktime + b

plt.plot(s.t,s.firstharmonic)
plt.plot(peaktime,xl)
plt.plot(time[peakindex],amplitude[peakindex],'o')
plt.show
```

After running the code we find the best fitted line.



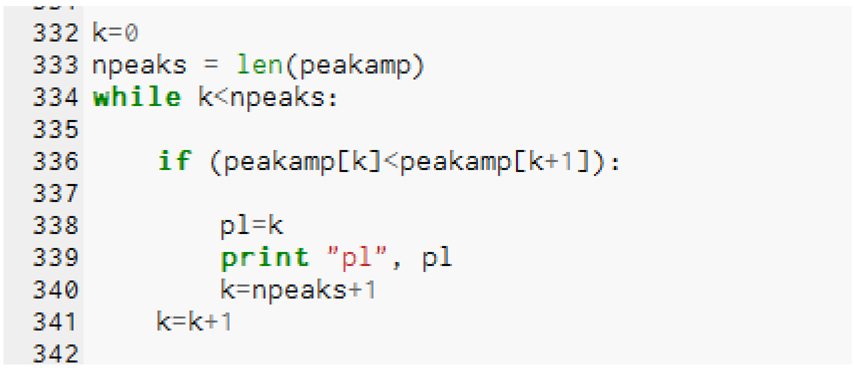
However, for calculating damping rate we need to draw a line through some first dots and not drow through noise level.

Because first dots show the amplitude of first harmonics, but next dots represent noise level.

Here is a code to find when noise starts.

If a amplitude is bigger than previous, it means that decay finishes and a noise started.

Actually it calculating a gradient. When the gradient becomes positive, it means a noise starts.

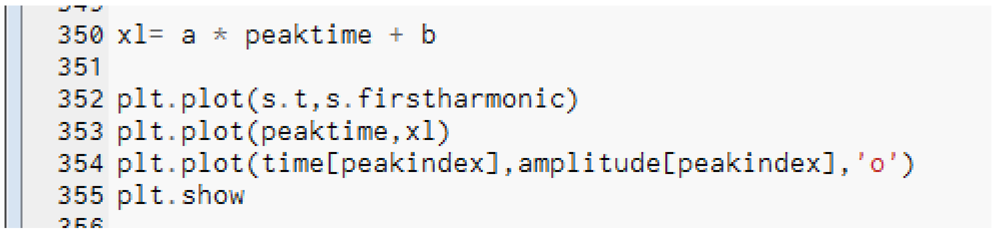


```
k=0
npeaks = len(peakamp)
while k<npeaks:

    if (peakamp[k]<peakamp[k+1]):

        pl=k
        print "pl", pl
        k=npeaks+1
    k=k+1

```

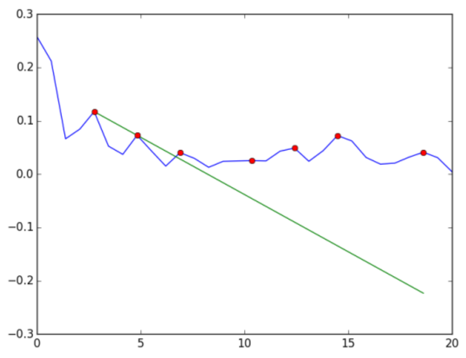
pl is number of peak when noise starts. So now we will draw a line before the noise level,

it shows the level of decay of first harmonic, or damping rate.

Here we found confidents of line a= popt[0] and b =popt[1], (y=a*x+b) and now will print the line



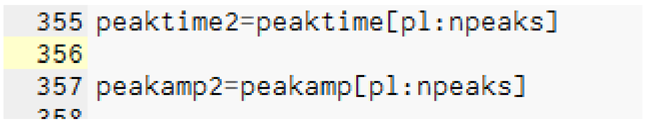
```
xl= a * peaktime + b

plt.plot(s.t,s.firstharmonic)
plt.plot(peaktime,xl)
plt.plot(time[peakindex],amplitude[peakindex],'o')
plt.show
```



Here value a= popt[0] is the damping rate, because it represent the slope of the line.

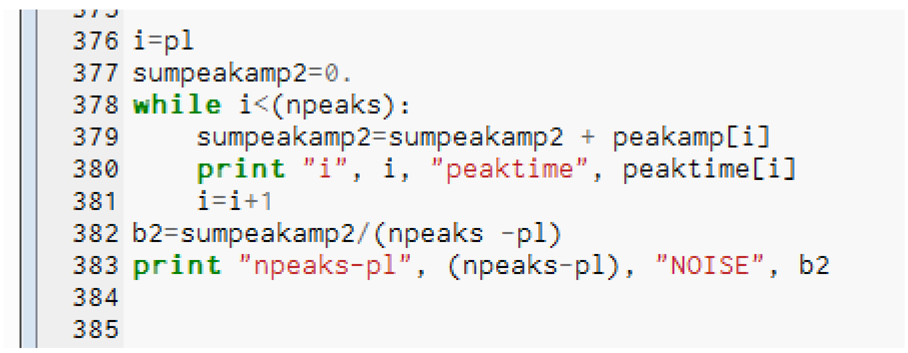
CALCULATING THE NOOISE LEVEL.

For calculating noise level we need to consider the rest of peaks and ignore peaks representing damping rate. We know the number of peak “pl” when damping finish.



```
peaktime2=peaktime[pl:npeaks]

peakamp2=peakamp[pl:npeaks]
```

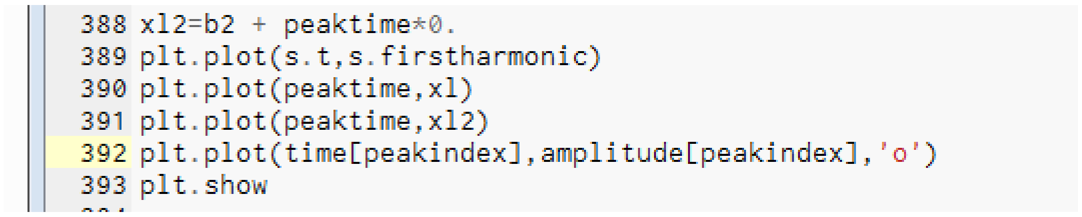
So, here we again we will use “argrelmax” python tools.



```
i=pl
sumpeakamp2=0.
while i<(npeaks):
    sumpeakamp2=sumpeakamp2 + peakamp[i]
    print "i", i, "peaktime", peaktime[i]
    i=i+1
b2=sumpeakamp2/(npeaks -pl)
print "npeaks-pl", (npeaks-pl), "NOISE", b2
```

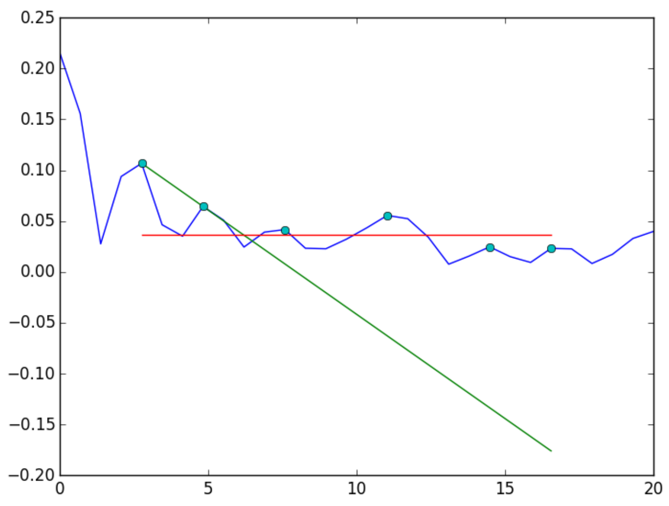
Here popt[1] = b2 represents a noise level.

Here a code for plotting a noise line.



```
xl2=b2 + peaktime*0.
plt.plot(s.t,s.firstharmonic)
plt.plot(peaktime,xl)
plt.plot(peaktime,xl2)
plt.plot(time[peakindex],amplitude[peakindex],'o')
plt.show
```

Here we get a plot, the red line is the level of noise.



ORGANISING A LOOP FOR THE CODE.

As we see it is very difficult to find precise value of noise and damping rate, because we don't have enough peaks. We need more statistics. For this we need to a few simulation of experiment.

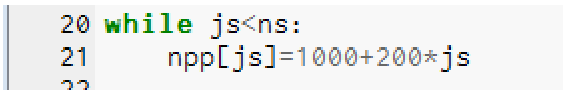
I decided to put the code in loop and run the code "ns" times.

For this we need to define some variables. I used python tool "zeros" and organise a loop

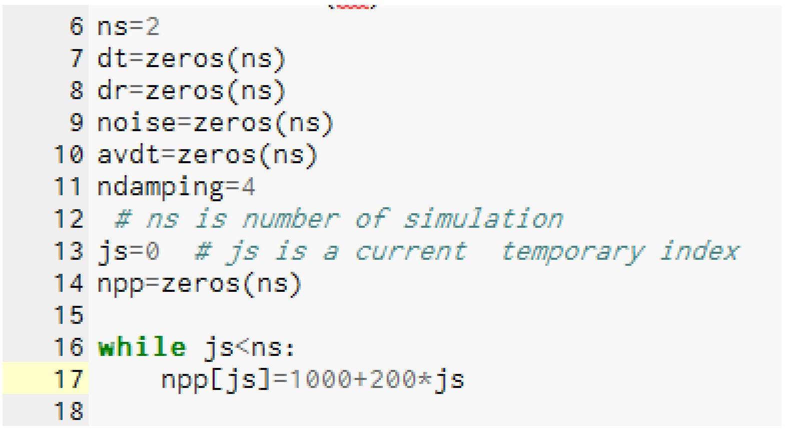
Using python tool "while ",

Defing number of running or number of running simulation as ns=2.

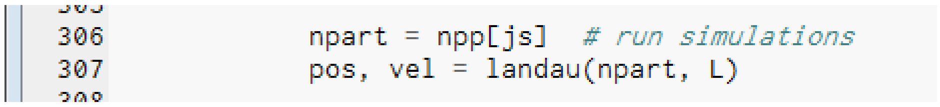
Changing inside the core code number of particles as npp(js) = 1000+200*js



```
while js<ns:
    npp[js]=1000+200*js
```

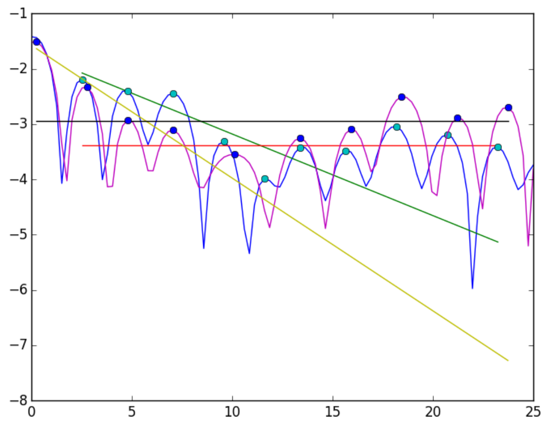


```
ns=2
dt=zeros(ns)
dr=zeros(ns)
noise=zeros(ns)
avdt=zeros(ns)
ndamping=4
 # ns is number of simulation
js=0   # js is a current  temporary index
npp=zeros(ns)

while js<ns:
    npp[js]=1000+200*js

```



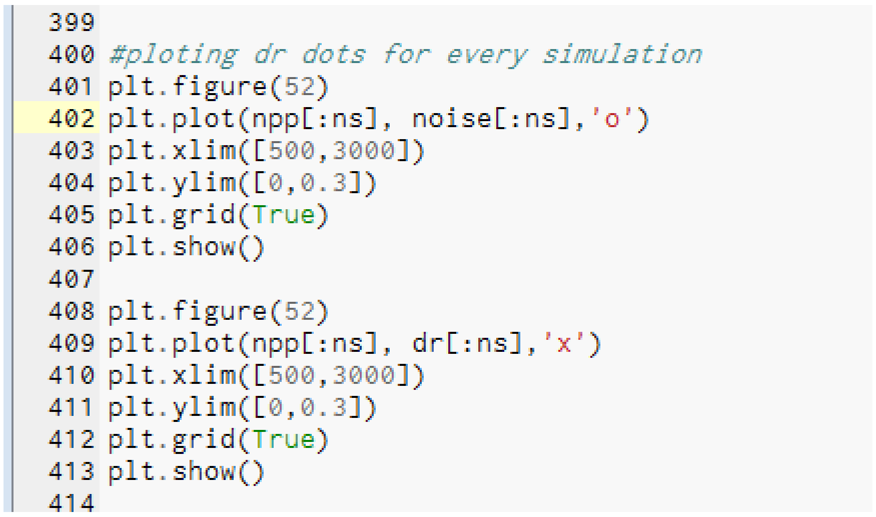
```
            npart = npp[js]  # run simulations
            pos, vel = landau(npart, L)
```

After running a few simulations I get this figure.

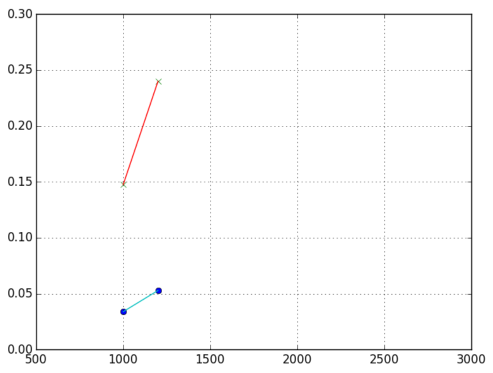


So we get a number of values of damping rates vs number of simulation or number of particles.

So we can print and plot them and calculate an average value with error.

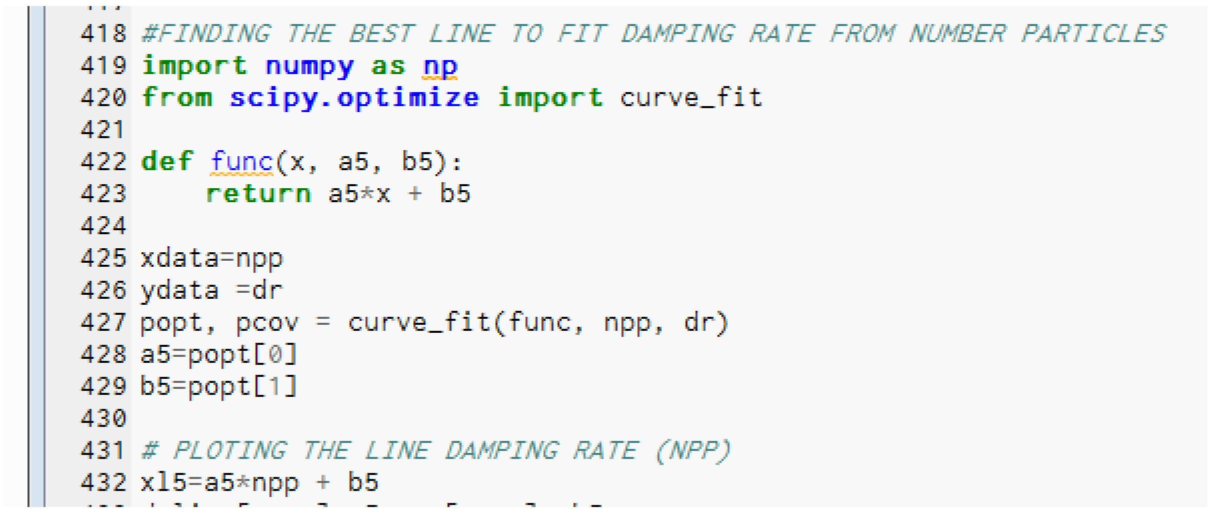


```
#ploting dr dots for every simulation
plt.figure(52)
plt.plot(npp[:ns], noise[:ns],'o')
plt.xlim([500,3000])
plt.ylim([0,0.3])
plt.grid(True)
plt.show()

plt.figure(52)
plt.plot(npp[:ns], dr[:ns],'x')
plt.xlim([500,3000])
plt.ylim([0,0.3])
plt.grid(True)
plt.show()
```



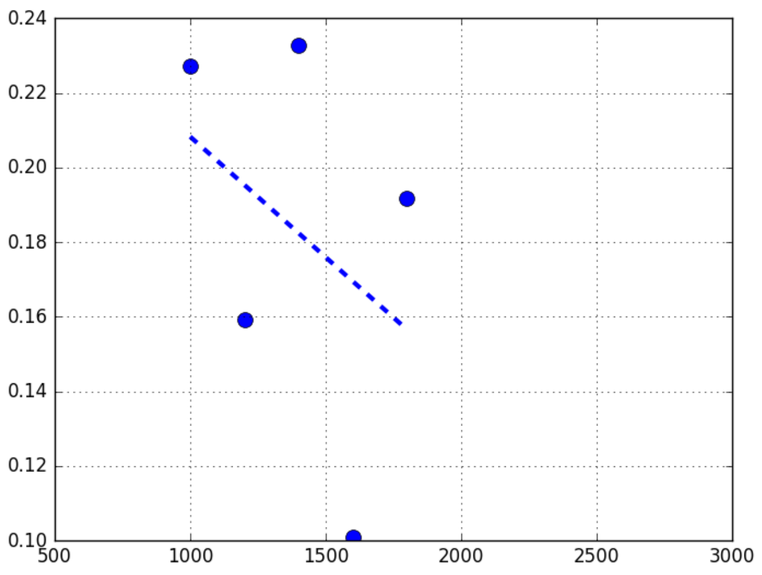
This figure above shows damping rate and level of noise vs number of particles.

Follow the similar procedure we can draw the best fitted line though dots representing damping rate vs number of particles.

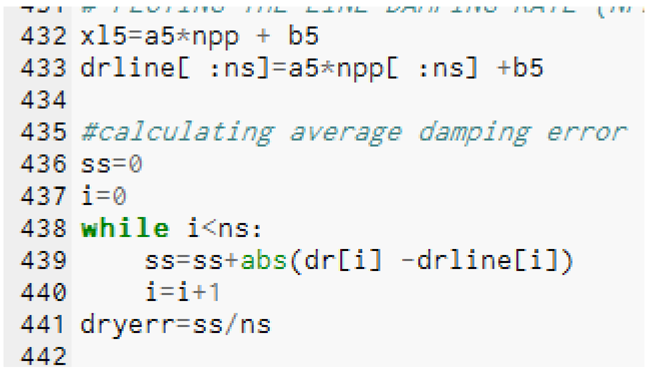


```
#FINDING THE BEST LINE TO FIT DAMPING RATE FROM NUMBER PARTICLES
import numpy as np
from scipy.optimize import curve_fit

def func(x, a5, b5):
    return a5*x + b5

xdata=npp
ydata =dr
popt, pcov = curve_fit(func, npp, dr)
a5=popt[0]
b5=popt[1]

# PLOTING THE LINE DAMPING RATE (NPP)
xl5=a5*npp + b5
```

After the code running we get the figure

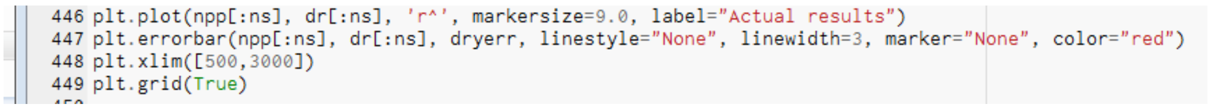


Calculating an error of damping rate. I wrote a code for this.



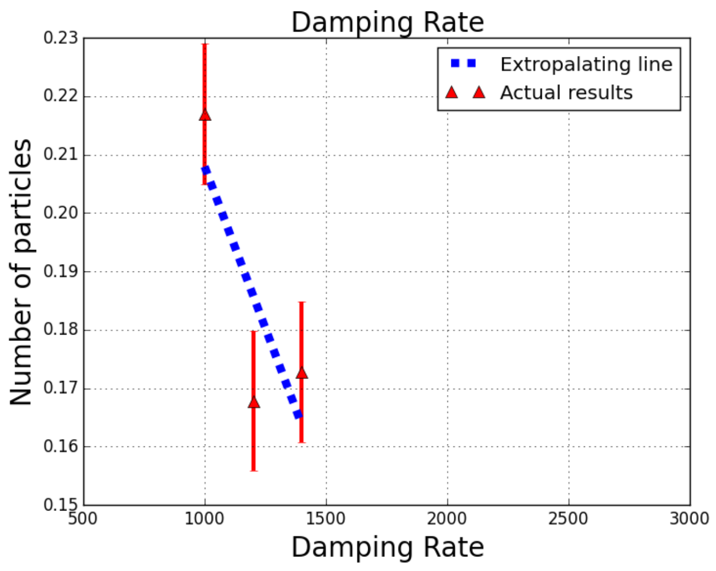
```
x15=a5*npp + b5
drline[ :ns]=a5*npp[ :ns] +b5

#calculating average damping error
ss=0
i=0
while i<ns:
    ss=ss+abs(dr[i] -drline[i])
    i=i+1
dryerr=ss/ns

```

And code for plotting this error of damping rate.

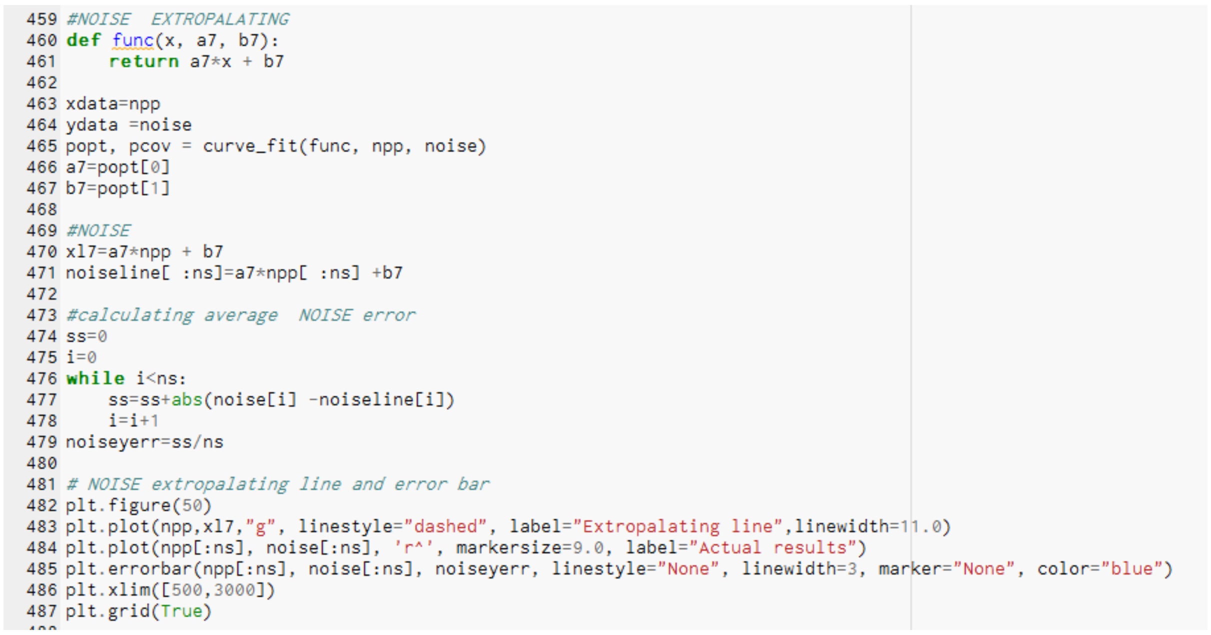


```
plt.plot(npp[:ns], dr[:ns], 'r^', markersize=9.0, label="Actual results")
plt.errorbar(npp[:ns], dr[:ns], dryerr, linestyle="None", linewidth=3, marker="None", color="red")
plt.xlim([500,3000])
plt.grid(True)
```

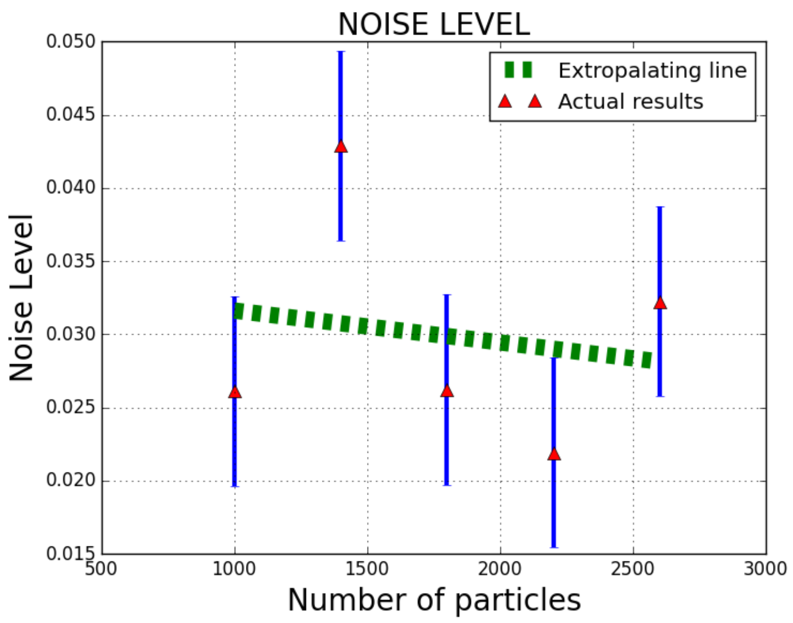
And I get a plot of damping rate vs number of particles with errorbar.



The similar procedure I do regarding to noise level and frequency.
I find the best fitted line for data noise,
Then calculated a error for every point, and found an average for all peaks.
Then I plotted the error bars with extrapolating line.

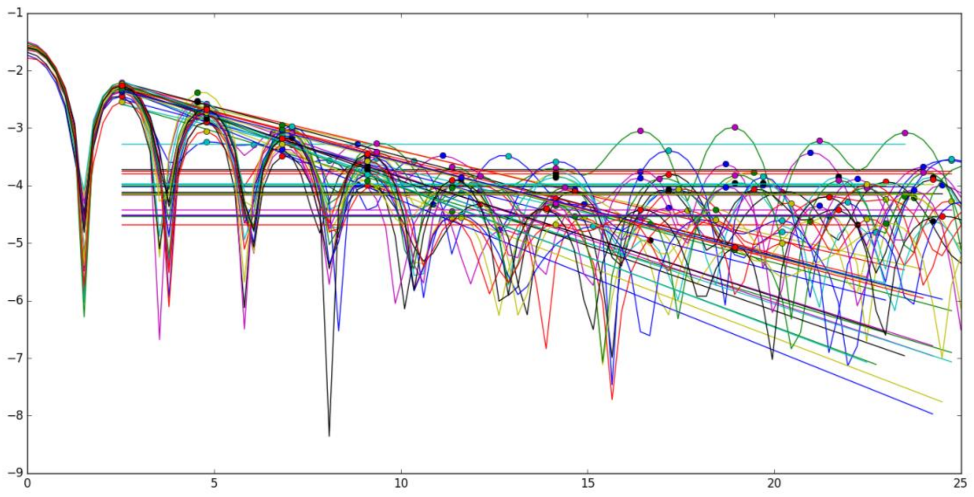


```python
#NOISE  EXTROPALATING
def func(x, a7, b7):
    return a7*x + b7

xdata=npp
ydata =noise
popt, pcov = curve_fit(func, npp, noise)
a7=popt[0]
b7=popt[1]

#NOISE
xl7=a7*npp + b7
noiseline[ :ns]=a7*npp[ :ns] +b7

#calculating average  NOISE error
ss=0
i=0
while i<ns:
    ss=ss+abs(noise[i] -noiseline[i])
    i=i+1
noiseyerr=ss/ns

# NOISE extropalating line and error bar
plt.figure(50)
plt.plot(npp,xl7,"g", linestyle="dashed", label="Extropalating line",linewidth=11.0)
plt.plot(npp[:ns], noise[:ns], 'r^', markersize=9.0, label="Actual results")
plt.errorbar(npp[:ns], noise[:ns], noiseyerr, linestyle="None", linewidth=3, marker="None", color="blue")
plt.xlim([500,3000])
plt.grid(True)
```



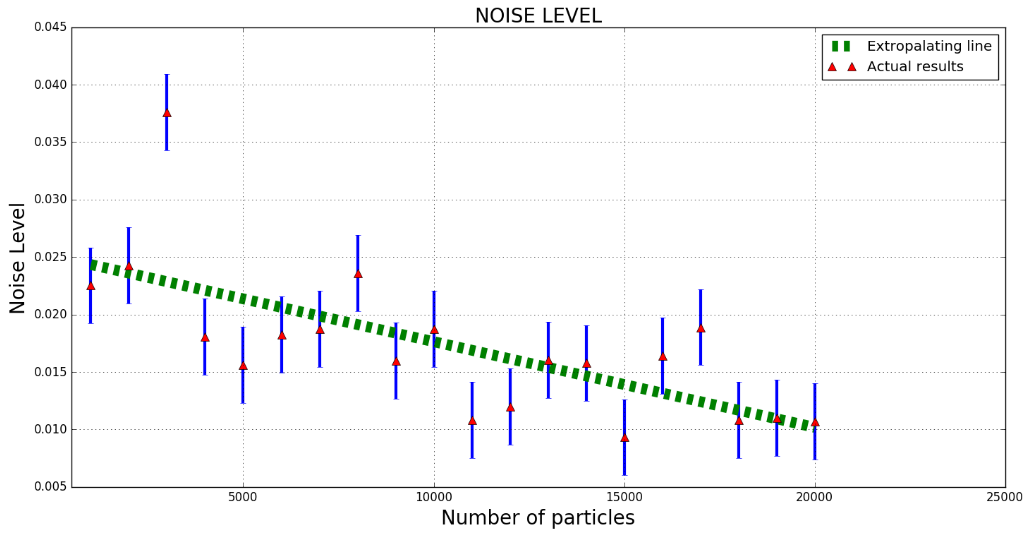
I perform absolutely the same procedure for analysing frequency of changing amplitude.
I just rename variables, for example I replace “dr” by “noise”, e tc.
By Now the code is tested by running only by few simulations (2, 3 or 5 simulations) Because it is time consuming to run code every time.
Approximately, every run of code takes 2 or 3 minutes. For writing and testing every peace of code For example for calculating level of noise takes 1, 2 or even 3 hours. Because even I make a tiny mistake the code is not working. So I write a little bit, 1 or 2 lines of code, and test it. So it is a very time consuming to write a long code.

However, when the code is written and tested, I have an opportunity to run the code with 20 simulations and analyse the data properly.

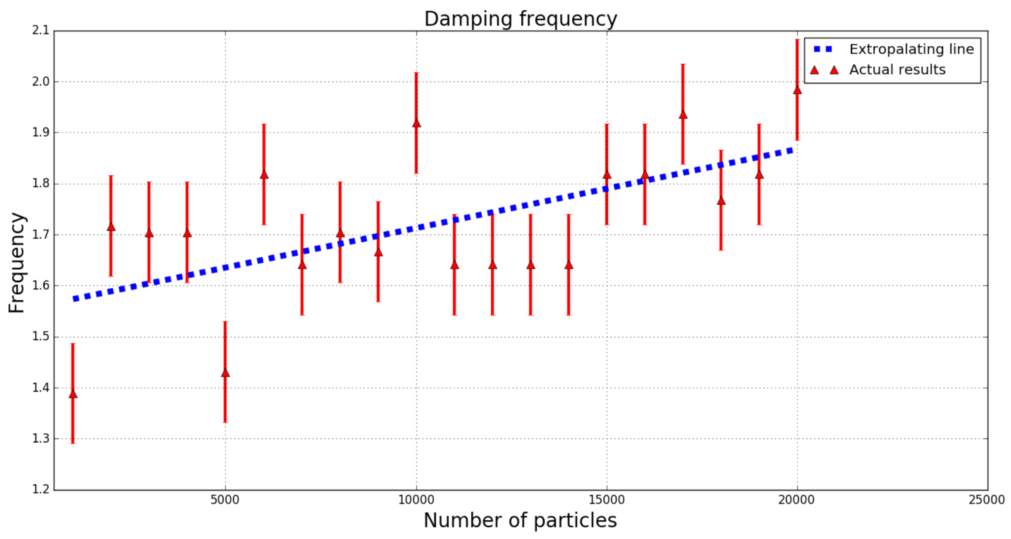


The figure above shows running code 20 times and measuring damping rate, level of noise and frequency.

The figure above shows damping rate vs number of particles.



The figure above shows level of noise vs number of particles. From this figure it is clear that level of noise is decreasing with increasing number of particles.

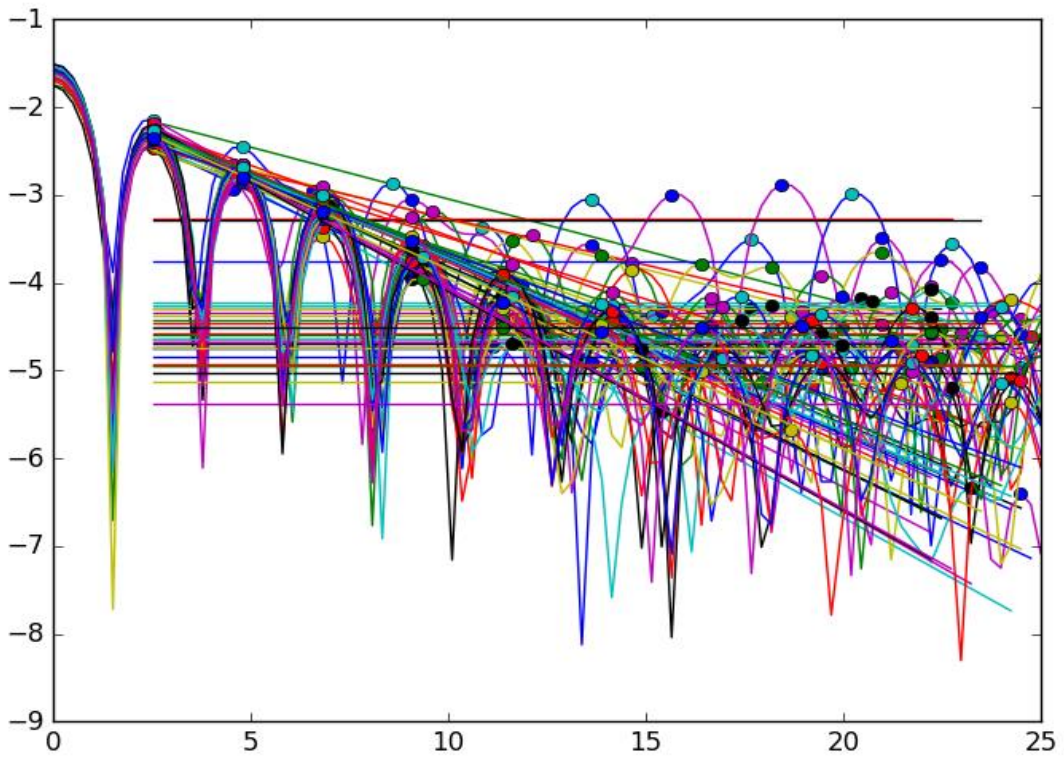


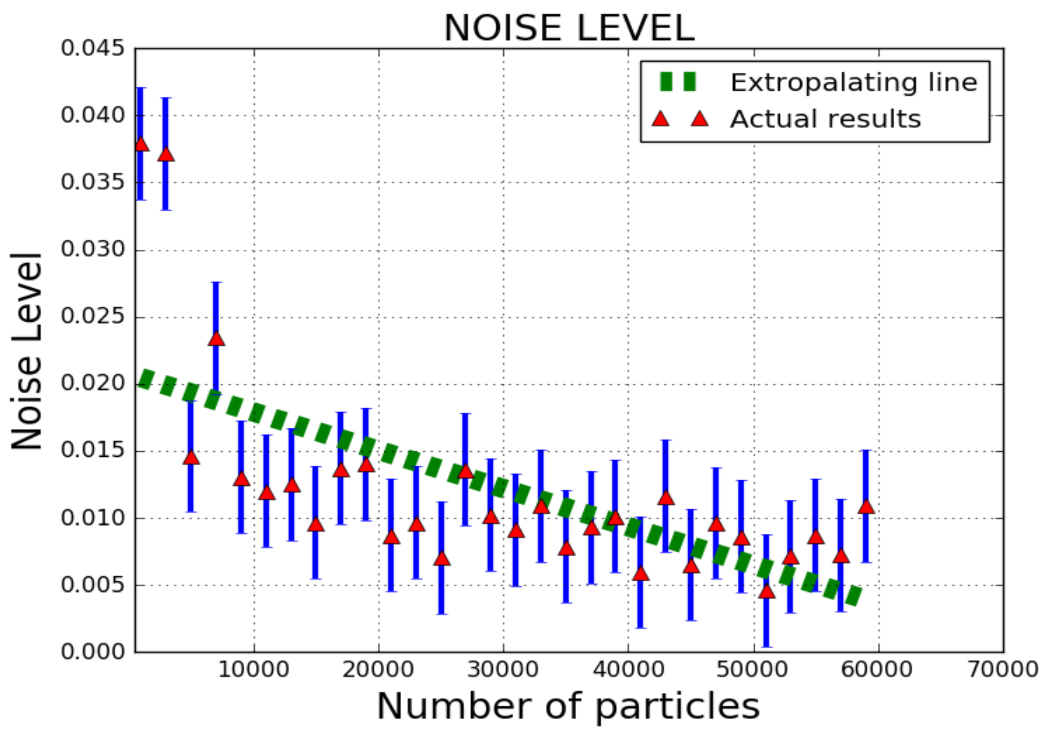
This figure should represent frequency, however it shows time interval between peaks. It is a mistake. We should calculate frequency as Frequency = 2* pi/(2* timeinterval)

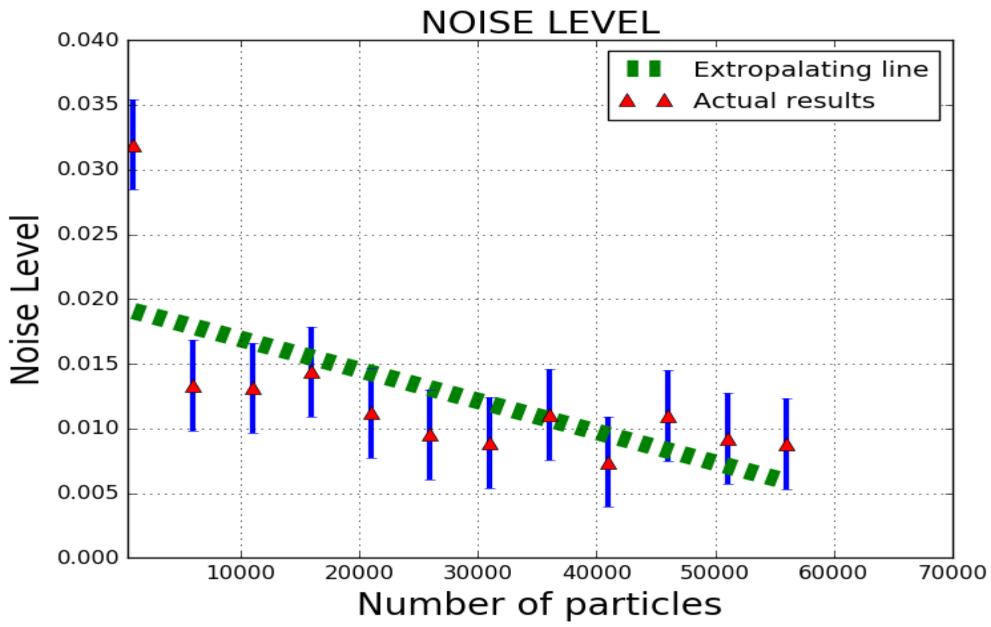
So frequency = 3.14/1.7= 1.8 (rad/sec)

Running multiple simulations with big number of particles.

It is very time consuming, because for example, running this simulation took 2 hours.Here are results.



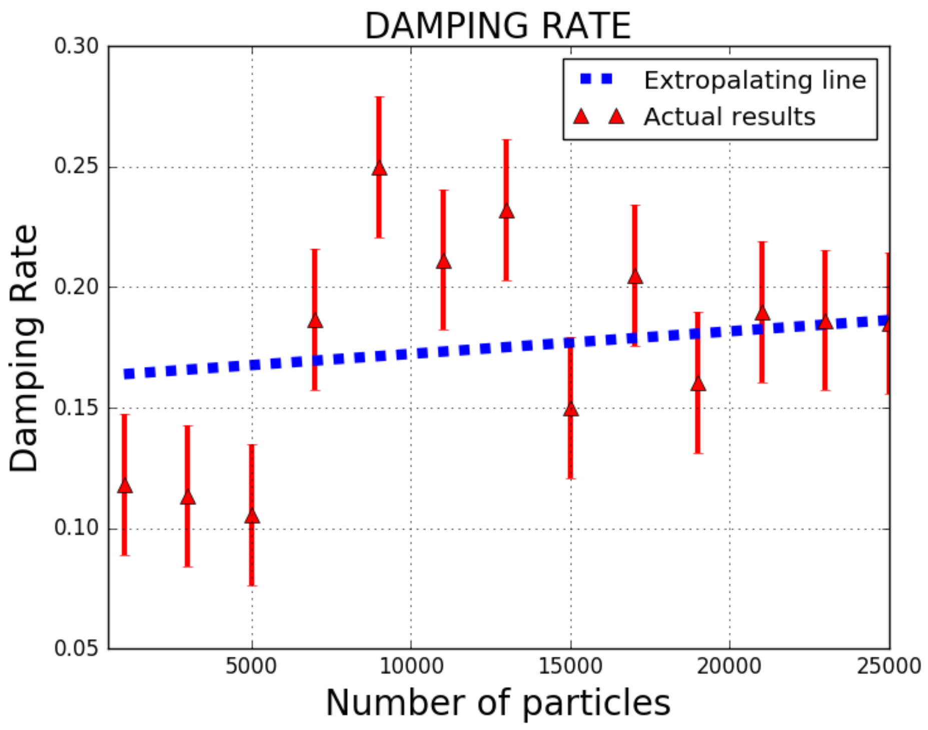




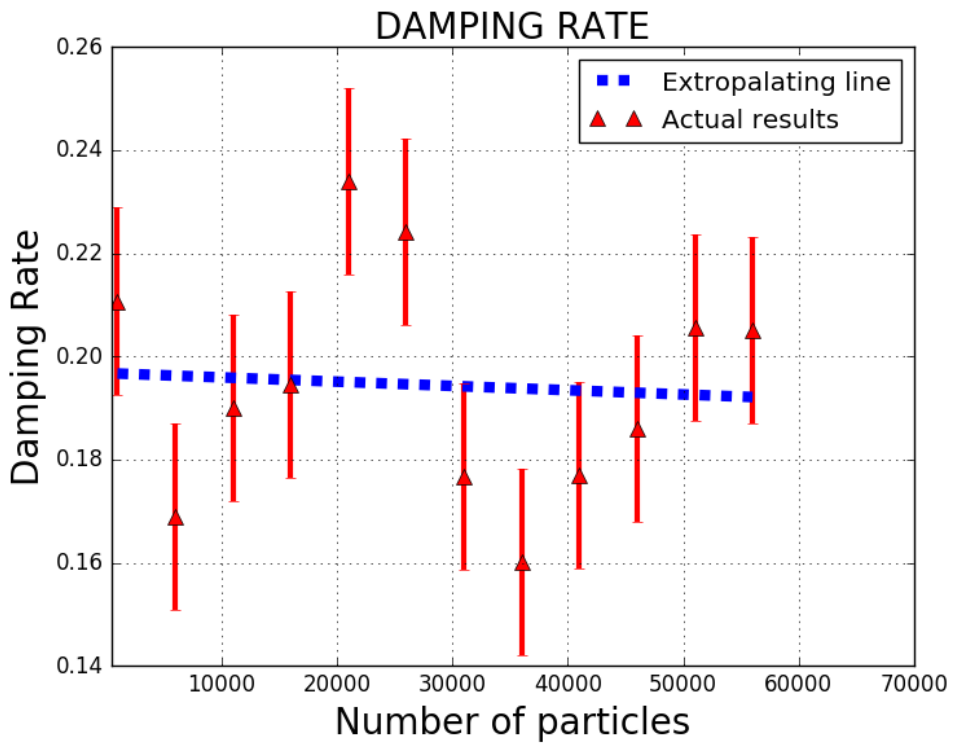
It is absolutely clear that level of noise decreases with increasing number of particles and

Has range from 0.06 to 0.09.

The situation with damping rate is not clear enough.

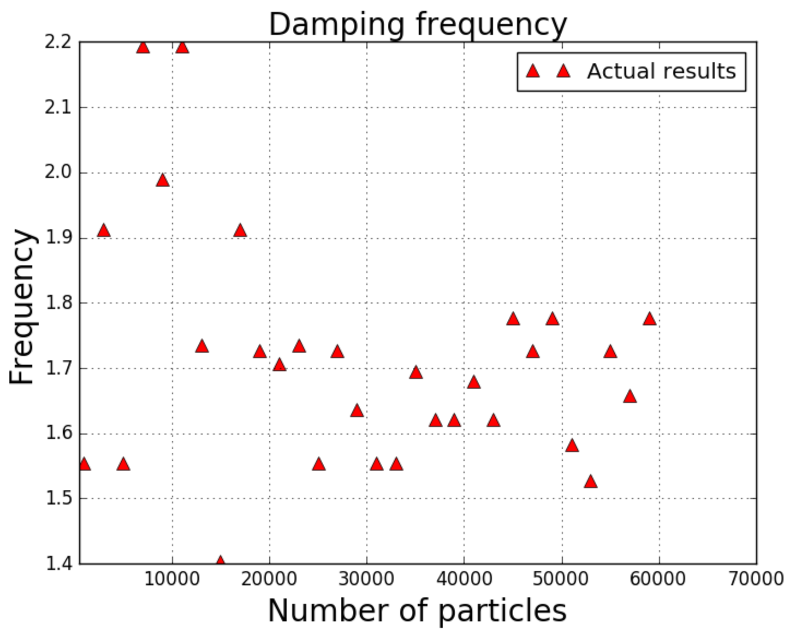


The figure above shows that damping rate is slightly increasing with increasing number of particles. However the next result shows something different.



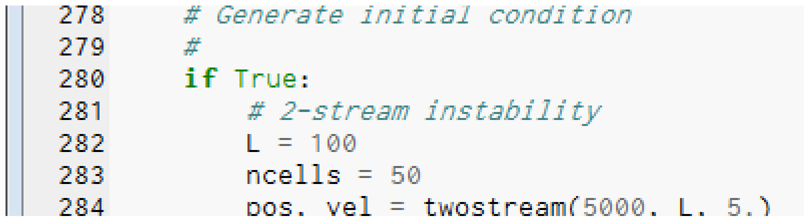
Even though damping rate has the same value, which is equal 0.17 -0.18., the damping rate is decreasing.

The situation with damping frequency is not clear at all and really confusing.
It look like this value doesn't correlate with number of particles at all. Here is the result.



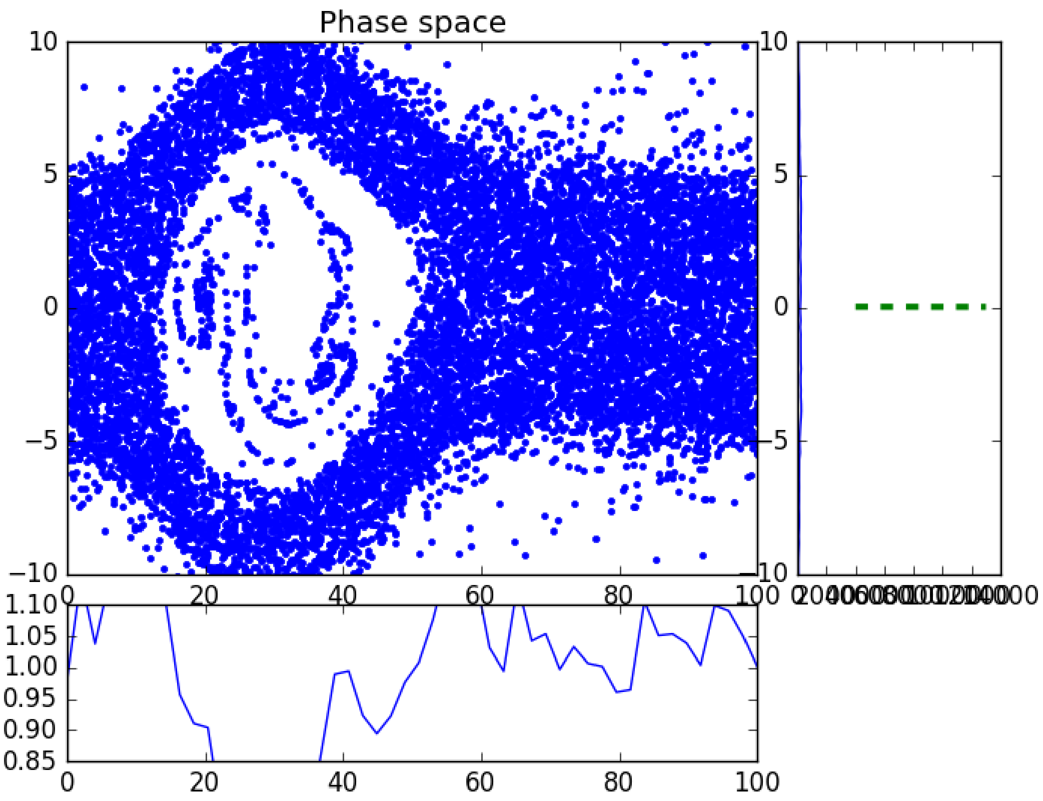
LAB 3, TWO STREAM INSTABILITY

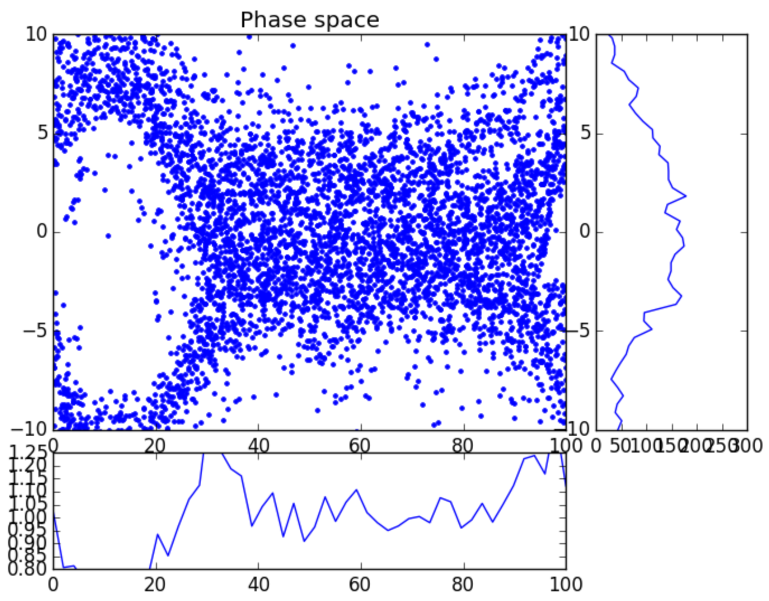
If make some changes in the original code –epcld.py, the code will simulate two stream instability.



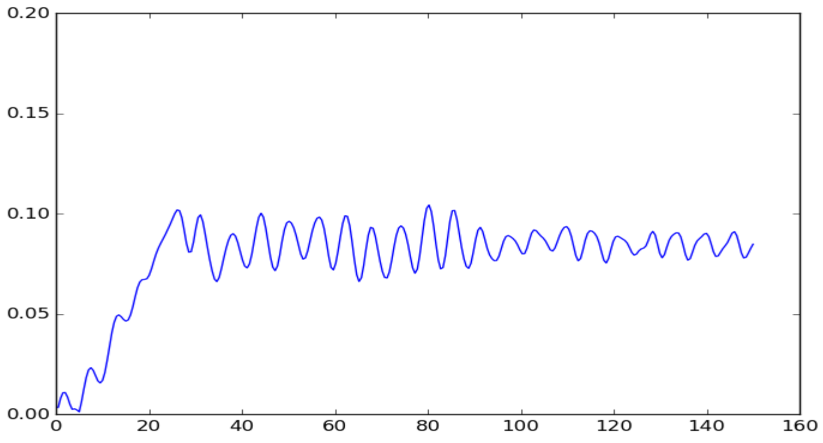
```
    # Generate initial condition
    #
    if True:
        # 2-stream instability
        L = 100
        ncells = 50
        pos, vel = twostream(5000, L, 5.)
```

The code will produce following animation :





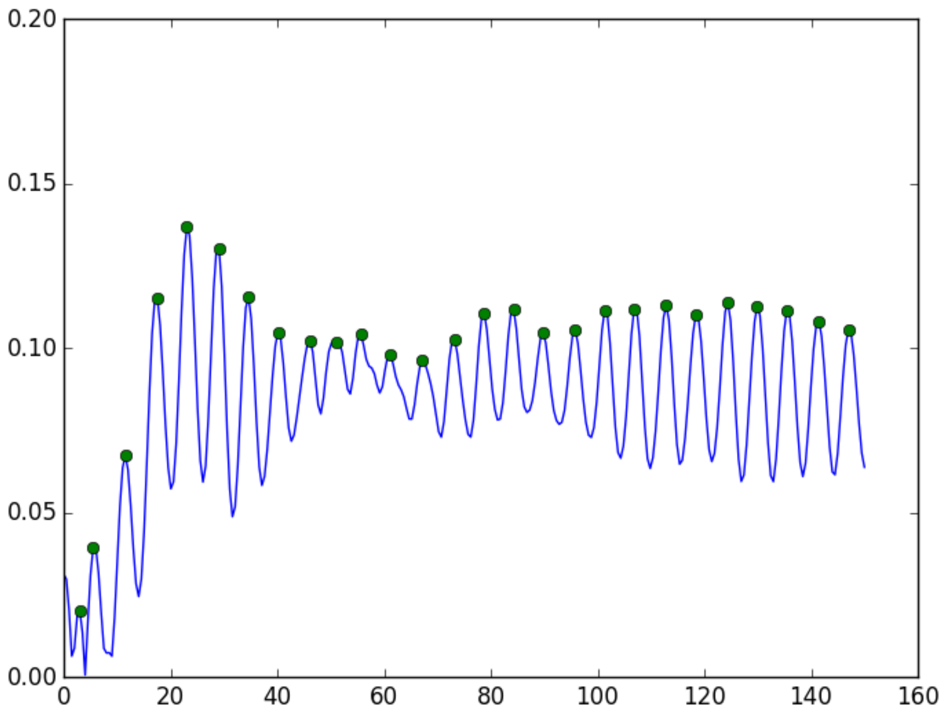
And show the wave increase vs time, it looks differently comparing with previous lab simulation. The amplitude is increasing instead up to some of level.
So damping rate is a positive number.



The task in this lab is very similar to previous, we need to measure a level of damping, increasing first harmonic amplitude. So, find peaks of the wave. For this we write the similar to previous lab python code.

```
amplitude = array(s.firstharmonic)
time = array(s.t)
peakindex = argrelmax(amplitude)[0]
peakindex = argrelmax(amplitude, axis=0, order=2)

peaktime = time[peakindex]
peakamp  = amplitude[peakindex]

```

And we get the peaks, which are represented by green dots.



As we see from the figure above 5 first dots represents increasing amplitude, and then the amplitude come to the some level. So we need to find and calculate the rate of increasing amplitude at 5 first peaks. But the biggest problem is number of first dots when amplitude increasing can be not 5, but any number, for example 3, 7, 10 etc. We don't know. But we would like to write a code which will do this automatically.

Also level of the peaks is unpredictable, the peaks jump up and down. It would be extreamally difficult to implement previos method and calculate a gradient

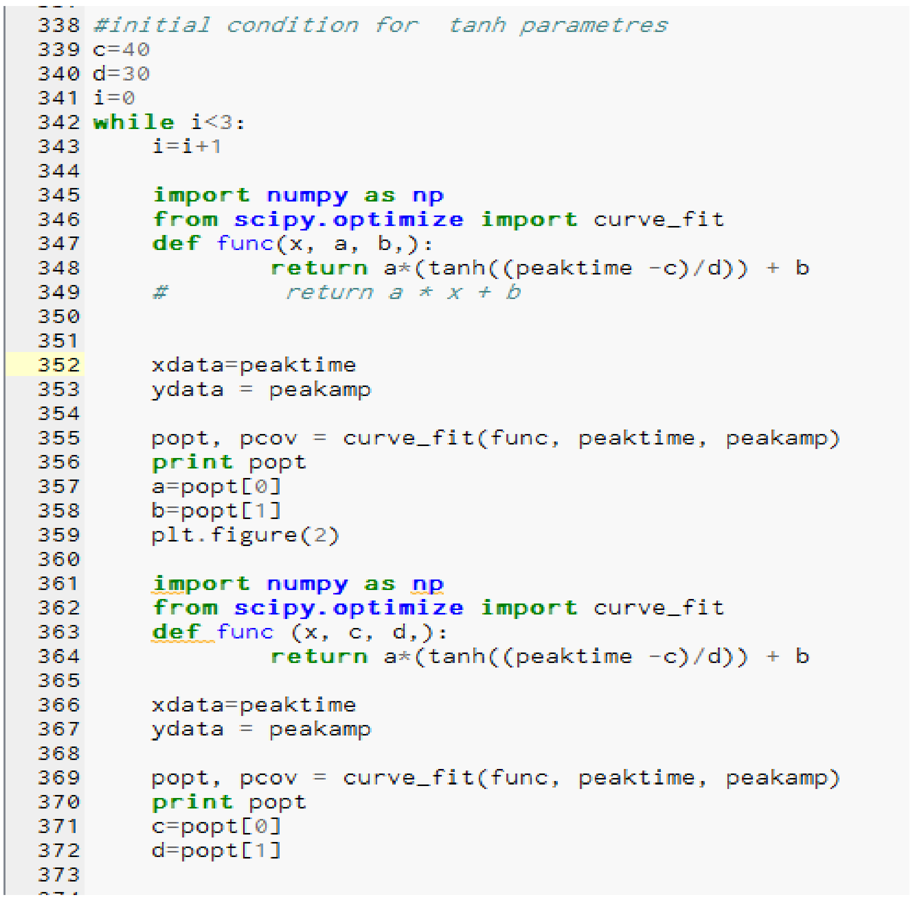
I suugest a different aproach to problem. I will try to exropalate these peaks with some function and I think that the function y=tanh(x) can be the best for this purpose.

The function tanh(x) has 4 parametres a, b, c and d. y(x)= a +b*tanh((x-c)/d).

I tried to use python rutine, the similar to previous lab, but the code doesn't work.

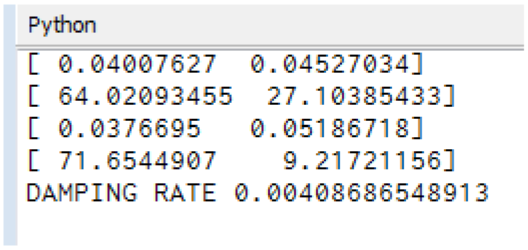
I desided to work out this problem step by step.

1) I defined initial parametres a and be and asked the code to calculate rest of parametres c and d to the best fit.
2) Once the code calculated c and d parametres of tanh(x), I asked the code to adjust a and b.
3) Then I put this procedure in loop, and calculated 4 tanh (x) parametres few times. Here is the code.



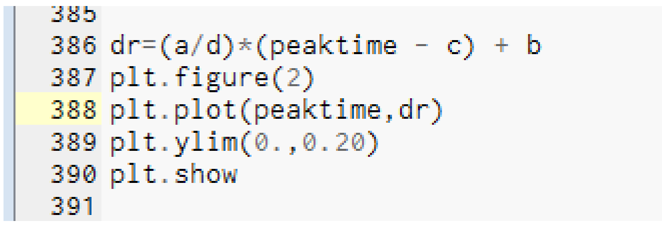
```
#initial condition for  tanh parametres
c=40
d=30
i=0
while i<3:
    i=i+1

    import numpy as np
    from scipy.optimize import curve_fit
    def func(x, a, b,):
            return a*(tanh((peaktime -c)/d)) + b
    #        return a * x + b


    xdata=peaktime
    ydata = peakamp

    popt, pcov = curve_fit(func, peaktime, peakamp)
    print popt
    a=popt[0]
    b=popt[1]
    plt.figure(2)

    import numpy as np
    from scipy.optimize import curve_fit
    def func (x, c, d,):
            return a*(tanh((peaktime -c)/d)) + b

    xdata=peaktime
    ydata = peakamp

    popt, pcov = curve_fit(func, peaktime, peakamp)
    print popt
    c=popt[0]
    d=popt[1]

```

So I got value for damping rate.



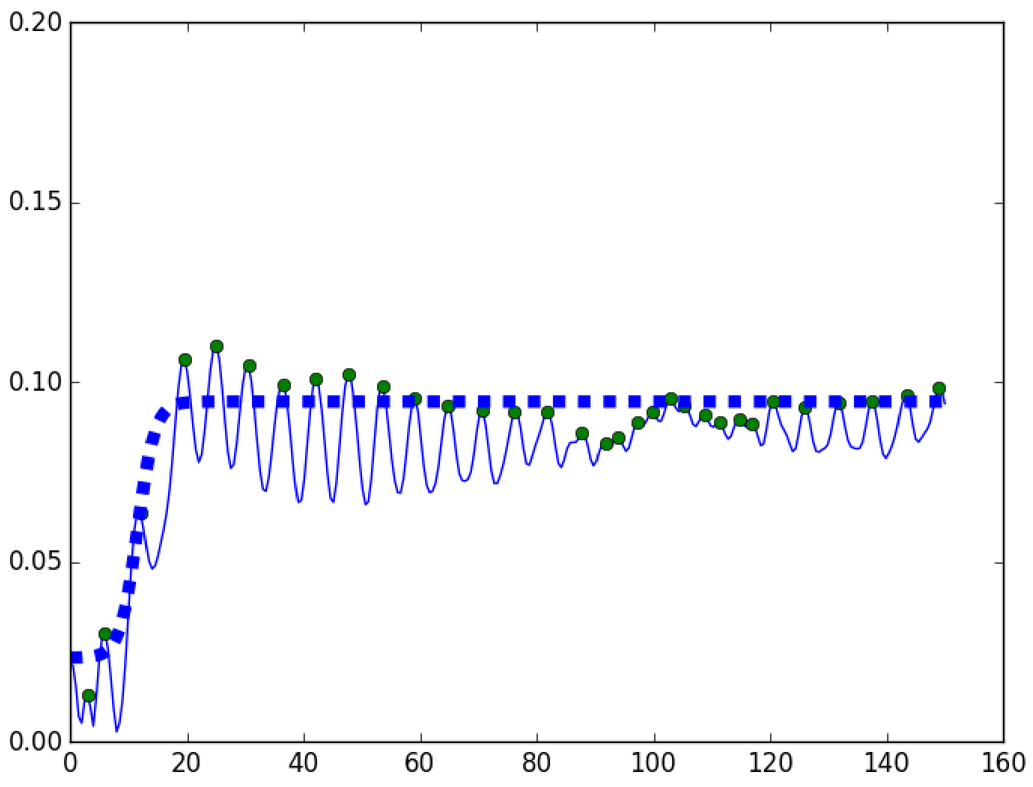
```
Python
[ 0.04007627  0.04527034]
[ 64.02093455  27.10385433]
[ 0.0376695   0.05186718]
[ 71.6544907    9.21721156]
DAMPING RATE 0.00408686548913
```

Knowing parametres a, b, c and d, I define an extropolating line

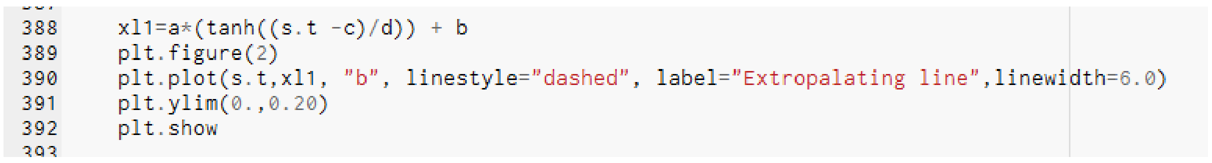


```
dr=(a/d)*(peaktime - c) + b
plt.figure(2)
plt.plot(peaktime,dr)
plt.ylim(0.,0.20)
plt.show
```

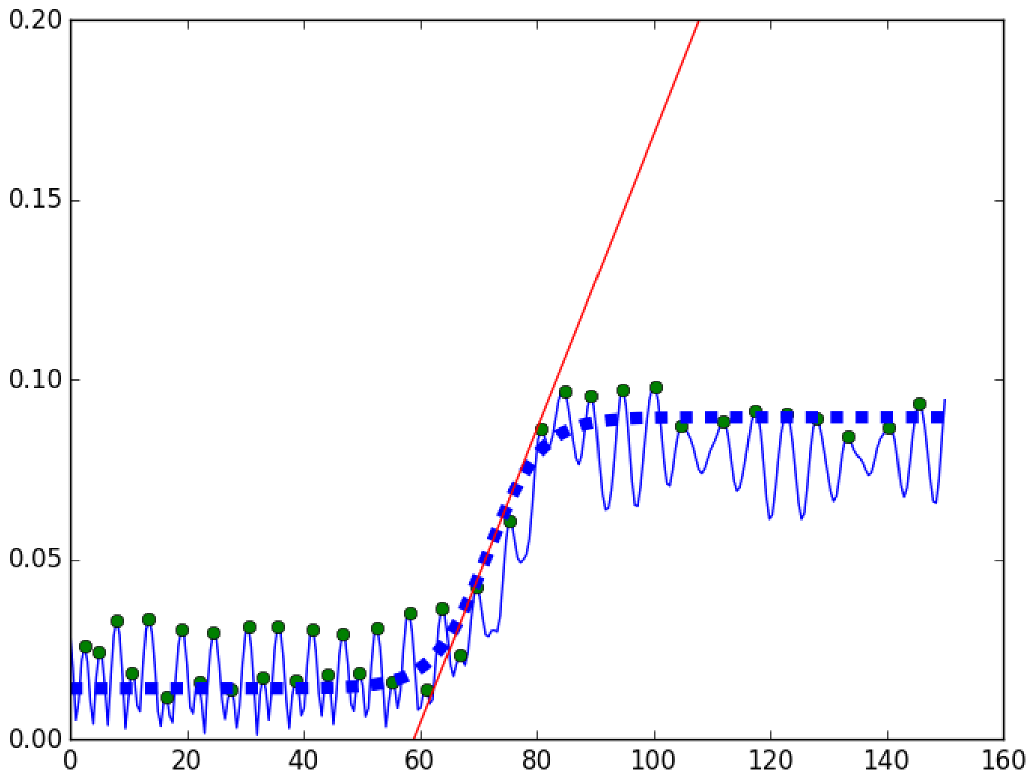
After running the code, I find a pretty good extropolation line



Now we need to find a gradiend of the extropalating line.



```
    xl1=a*(tanh((s.t -c)/d)) + b
    plt.figure(2)
    plt.plot(s.t,xl1, "b", linestyle="dashed", label="Extropalating line",linewidth=6.0)
    plt.ylim(0.,0.20)
    plt.show
```



The figure abow shows incresing amplitude of the first harmonic – blue line, its peaks – green dots, extropalating line - blue dashed line and gradiend, the rate of increase of amplitude of the first harmonic.

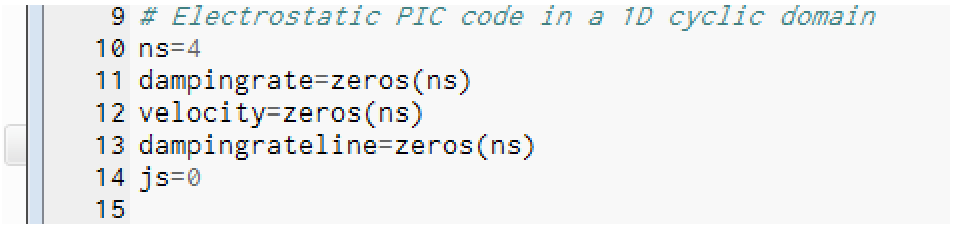
The biggest problem in analysing the data is peaks dots jump up and down, Sometimes the line has too many local peaks, which make difficult to analyse. We need to select the main maximus and ignore local naximus, For this purpose I use such a python tool as "order= " "". I use order = 3 or 5, depending on the situation.

I thought about weather it is possible the code find the parrametre of "order" itself. For example if neighbour maximuses differ from each other too much, the code increase "order" and analyses the date again until it has a "nice" and smoth line of peaks. (But because of luck of time I didn' realise the idea.)

Organising a loop.

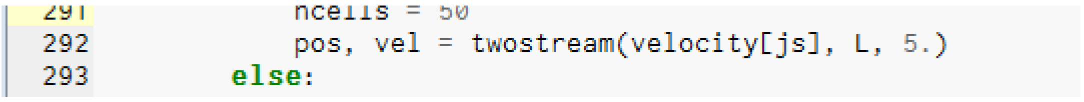
As we see from from simulations, results depents from number of particles, initial conditions and others parametres, and randomly generated numbers, te results have a big variations, and vary in 2 or 3 times. To get something sensible we need more statistics, and calculate average with error. So we need to run multiple simulations of experiment. So we need put the code in a loop.
For this I used the similar techniques as I did in Lab 2.
Here is a code for loop.

Here we define some variables,
Ns is number of simulations



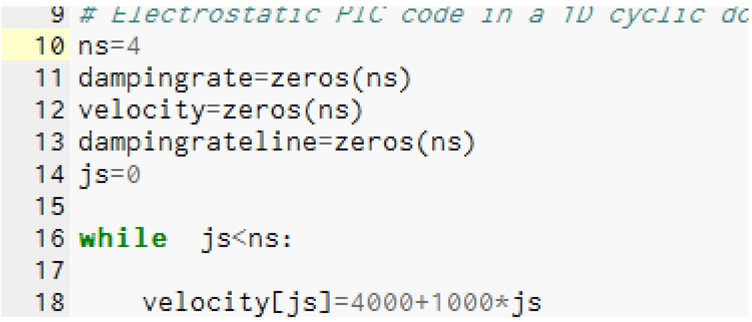
```
# Electrostatic PIC code in a 1D cyclic domain
ns=4
dampingrate=zeros(ns)
velocity=zeros(ns)
dampingrateline=zeros(ns)
js=0

```

Here we define velocity as a function from number of simulations



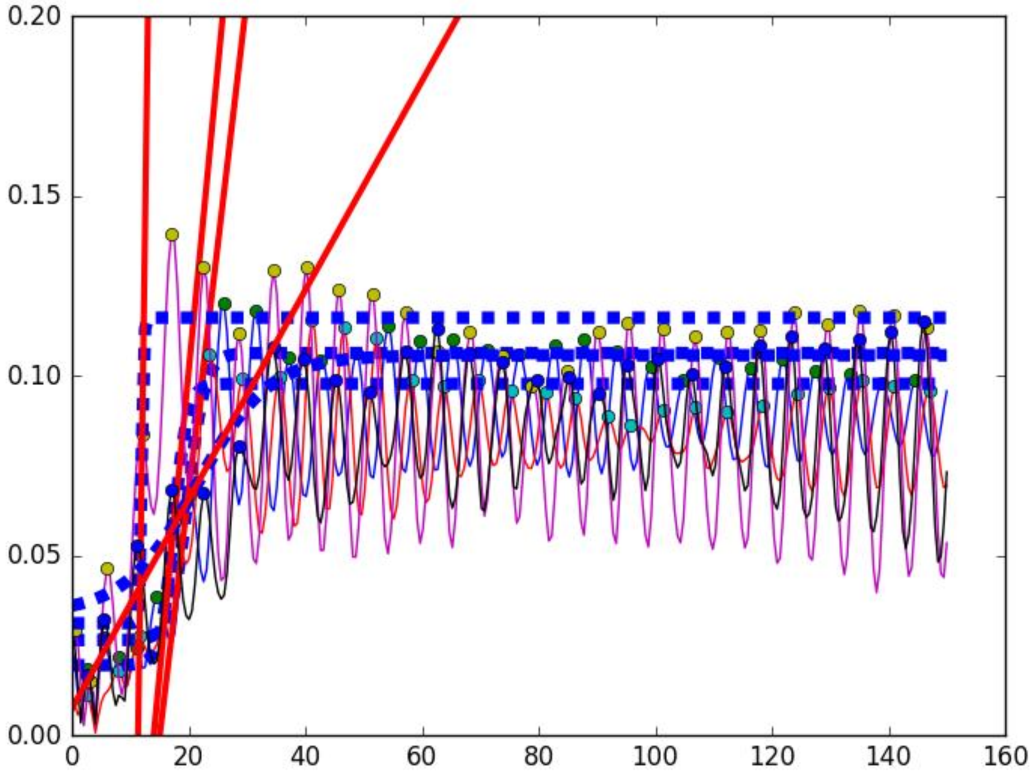
```
                ncells = 50
                pos, vel = twostream(velocity[js], L, 5.)
            else:
```

Here is the loop, I used python tool “while “



```
# Electrostatic PIC code in a 1D cyclic dc
ns=4
dampingrate=zeros(ns)
velocity=zeros(ns)
dampingrateline=zeros(ns)
js=0

while  js<ns:

    velocity[js]=4000+1000*js
```

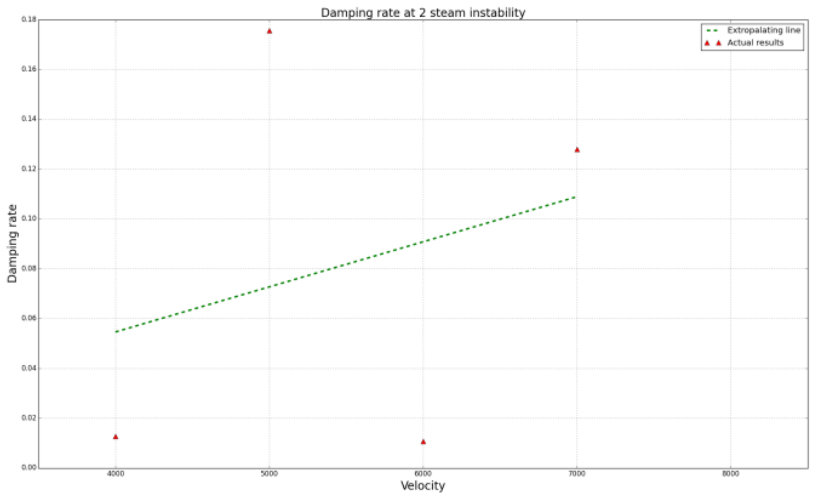
After running the code I got some figures.



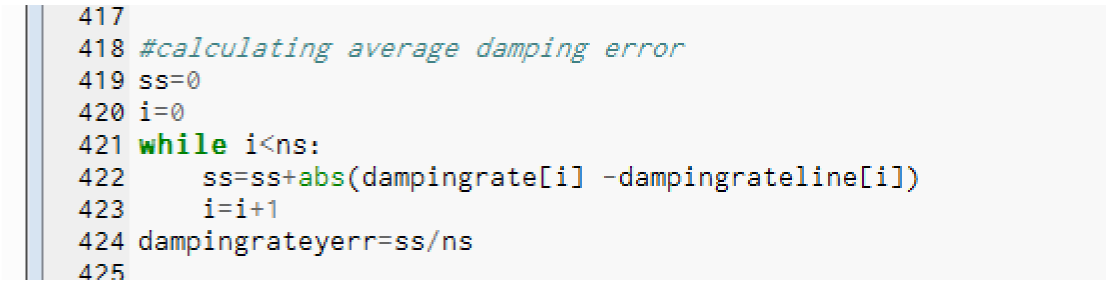
This figure shows damping rate represented by red line.

As we can see in some simulations gradient of extropalating line, the damping rate is unrealistically high. It happens because there are not enough peaks (dots) to fit extropolating line properly.

The damping rate has to be restricted by value = 2*b/(time between peaks).
If damping rate is higher than this value, the code should disregard the result and run the simulation again. (Because of limited time I didn't realise this my idea.)

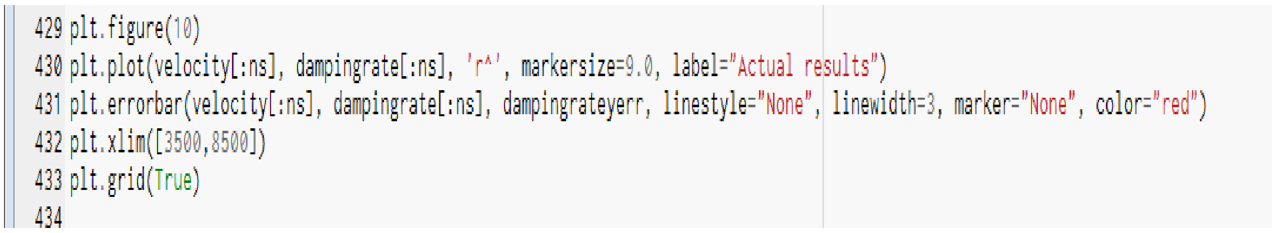


Here is a plot of damping rate vs velocity.
Using absolutely the same techniques as in lab 2, I will analyse error of damping rate.



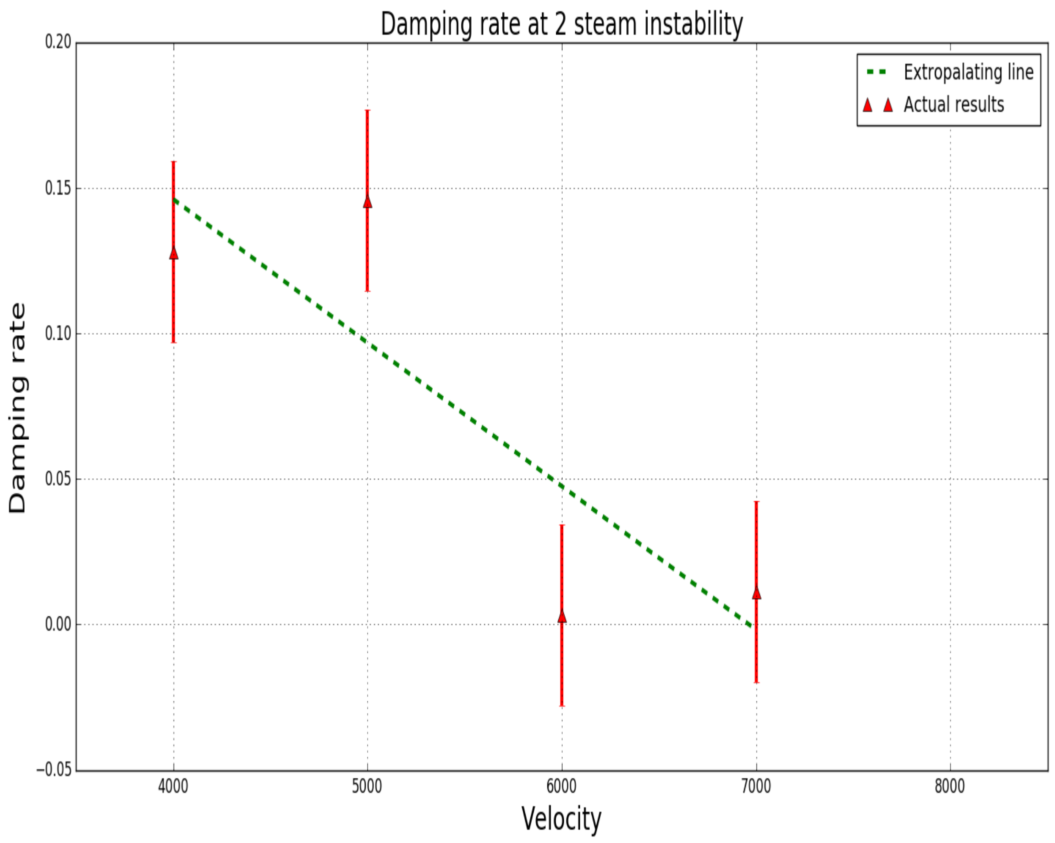
```
#calculating average damping error
ss=0
i=0
while i<ns:
    ss=ss+abs(dampingrate[i] -dampingrateline[i])
    i=i+1
dampingrateyerr=ss/ns
```

Here is a code for plotting a damping plot with error bar.

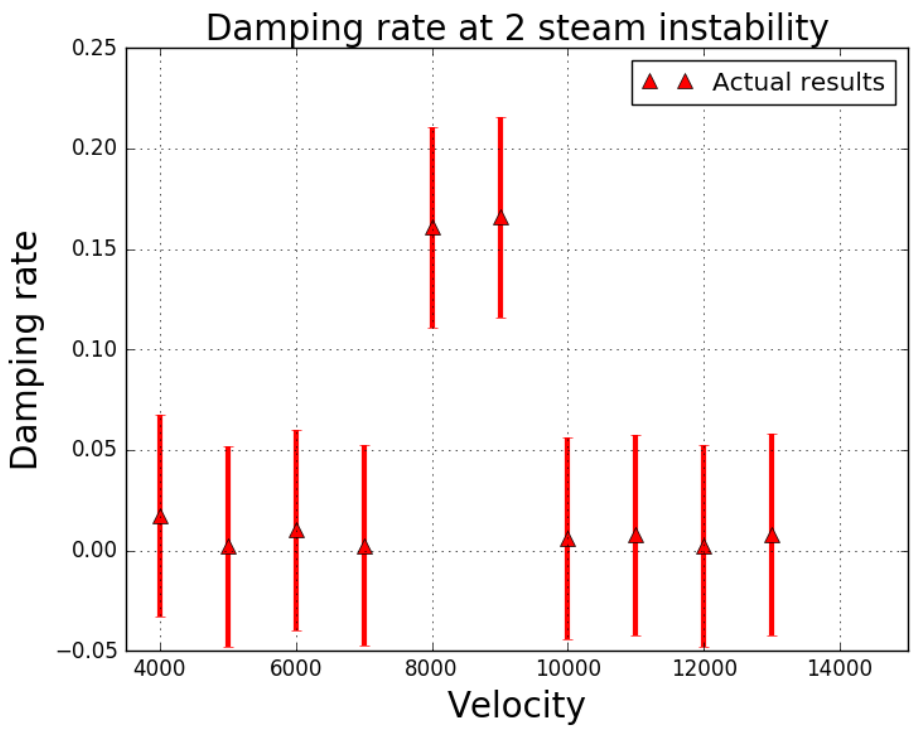


```
plt.figure(10)
plt.plot(velocity[:ns], dampingrate[:ns], 'r^', markersize=9.0, label="Actual results")
plt.errorbar(velocity[:ns], dampingrate[:ns], dampingrateyerr, linestyle="None", linewidth=3, marker="None", color="red")
plt.xlim([3500,8500])
plt.grid(True)
```

Here is the plot of damping rate vs   initial Velocity.



Here I run a few extra simulations, but some of them are not very succesful.



The code didn't draw the extropolating line.(To correct not "steam", but stream !!!!!)