`Python` introduction

NumPy Basics: Fitting and testing

`email=` ja.montana@uniandes.edu.co

12.08.2019

Our last example was based on how to manipulate data qualitatively, now we are going to start doing some statistics so that later there will be easier for us, do some analysis.

As we showed, `pandas` is a great alternative when doing data analysis. But sometimes there are some important libraries that uses different kind of data more *simple*.

So during this session we will use a `numpy` approach using `scipy`

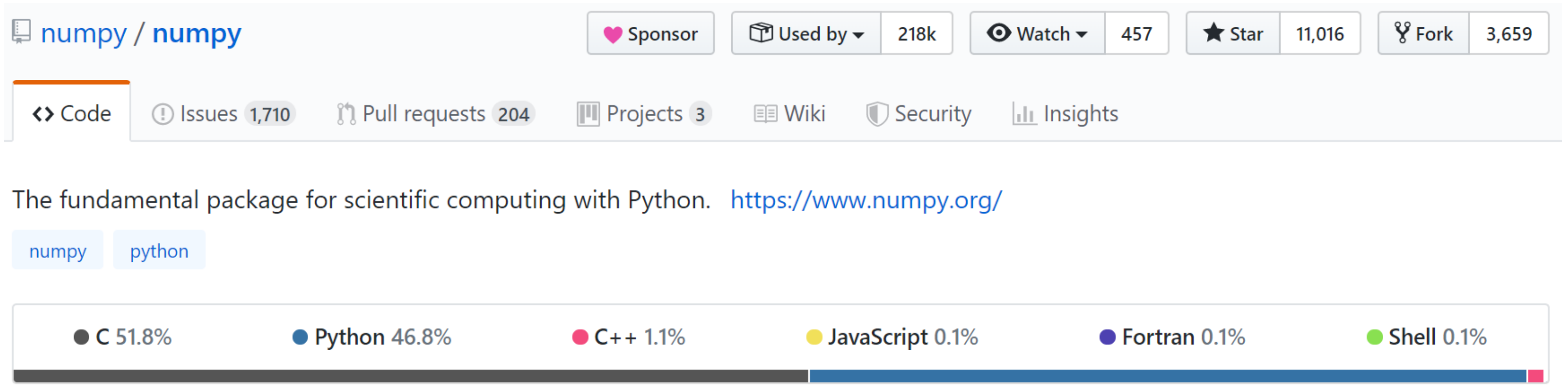
The interesting part here, is that `pandas` is based on `numpy` and `scipy`

Our main objective today, is getting some insides on `numpy` , data transformation and fitting.

By the moment, we will just need `numpy` and `matplotlib` ,

```
In [1]: import numpy as np
        import matplotlib.pylab as plt
```

`NumPy` brought many advantages to `Python` in terms of velocity of big calculations (One of the biggest problems of `Python`), we will see some examples and learn the usage meanwhile.



numpy / numpy

Sponsor | Used by 218k | Watch 457 | Star 11,016 | Fork 3,659

Code | Issues 1,710 | Pull requests 204 | Projects 3 | Wiki | Security | Insights

The fundamental package for scientific computing with Python. https://www.numpy.org/

numpy python

● C 51.8% ● Python 46.8% ● C++ 1.1% ● JavaScript 0.1% ● Fortran 0.1% ● Shell 0.1%

More than 50% is done on `C` !!

What makes `NumPy` special?

```
 - arrays!
```

Let us take two built-in lists,

```
In [2]: x=[1,2,3]
        y=[3,2,1]
```

For example we would like to sum them, component by component as vectors do

$$\begin{pmatrix}1\\2\\3\end{pmatrix}+\begin{pmatrix}3\\2\\1\end{pmatrix}=\begin{pmatrix}4\\4\\4\end{pmatrix}$$

In [3]:
```
z=x+y
```

In [4]:
```
print(z)
```

```
[1, 2, 3, 3, 2, 1]
```

`List` s do not work like that!, so we have to construct the sum from scratch.

In [5]:
```
z=[]
for i in range(len(x)):
    z.append(x[i]+y[i])
```

In [6]:
```
print(z)
```

```
[4, 4, 4]
```

There is a different way to do this but with some more structure.

```
In [7]: z=[x[i]+y[i] for i in range(len(x))]
```

```
In [8]: z
```

```
Out[8]: [4, 4, 4]
```

What if we want to use bigger `List` s, for example, using the `range` function

```
In [9]: x=list(range(10))
        print(x)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

But it has other useful parameters,

```
In [10]: x=list(range(1,10))
         print(x)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [11]: x=list(range(1,10,3))
         print(x)
```

```
[1, 4, 7]
```

Let us create a bigger list

In [12]:
```
N=10000000
x=list(range(N))
y=list(range(N))
```

And let us calculate the sum, but!, measuring the time it takes

```
In [13]: %%time
         z=[]
         for i in range(len(x)):
             z.append(x[i]+y[i])
```

```
Wall time: 2.41 s
```

```
In [14]: %%time
         z=[x[i]+y[i] for i in range(len(x))]
```

```
Wall time: 1.5 s
Compiler : 202 ms
```

These are called **magic** cells, done for the jupyter notebooks!, for more information use

```
%magic
```

on a code cell.

%magic

Now, using `array` s!

As we have been talling before, everytime we call a function from a given library (in this case `NumPy`), we use first the name (or alias.)

In [15]:
```
X=np.array(x)
Y=np.array(y)
```

There you can see that we *converted* from lists to arrays!!.

```
In [16]: %%time
         Z=X+Y

         Wall time: 25.7 ms

In [ ]:
```

The `array` s are created from lists or tuples, not numbers!!

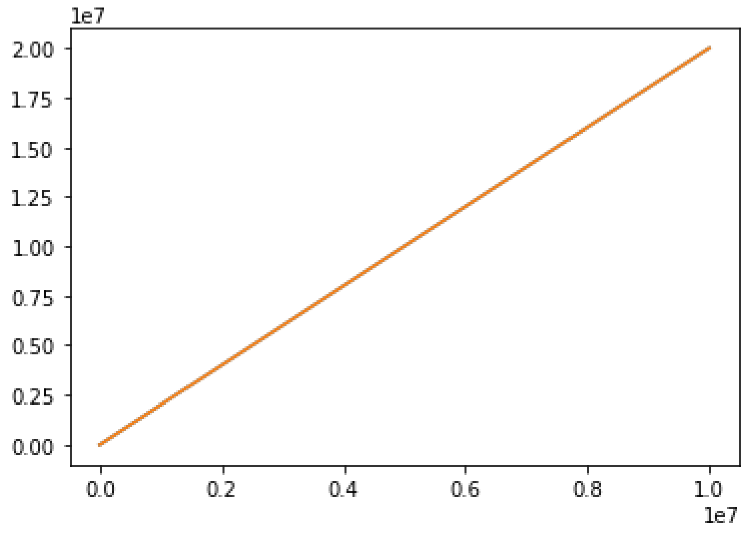
```
In [ ]:
```

How can we know if there is any difference between the results from `list` s and `array` s?

A: Plotting

In [17]:
```
plt.plot(Z)
plt.plot(z)
```

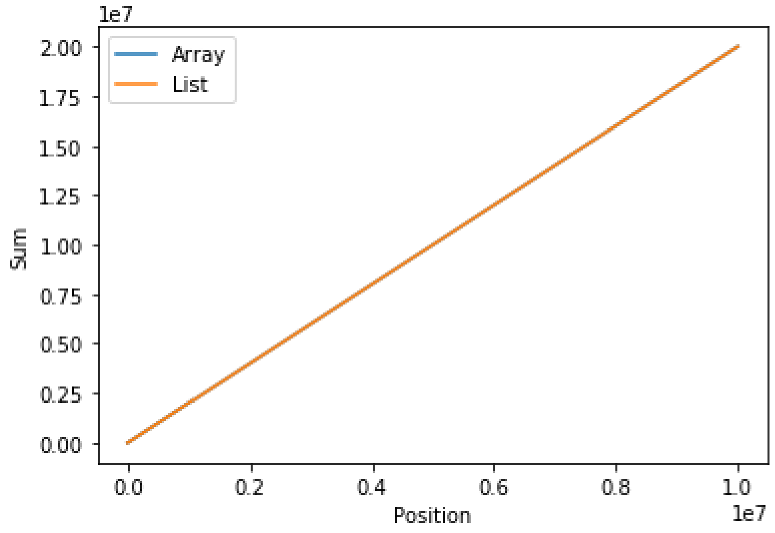
Out[17]:
```
[<matplotlib.lines.Line2D at 0x22c07755f98>]
```



That is not a presentable plot!

In [18]:
```
plt.plot(Z,label='Array')
plt.plot(z,label='List')
plt.xlabel("Position")
plt.ylabel("Sum")
plt.legend()
```

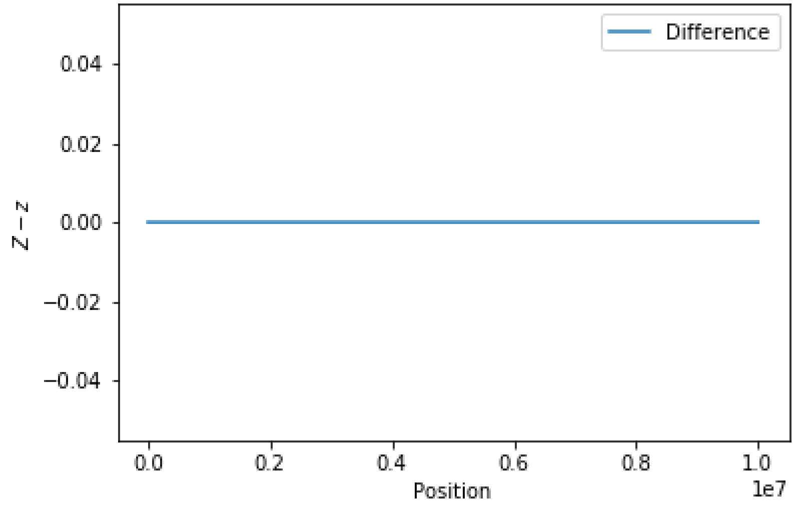
Out[18]:
```
<matplotlib.legend.Legend at 0x22c1c99a470>
```



Or the difference!

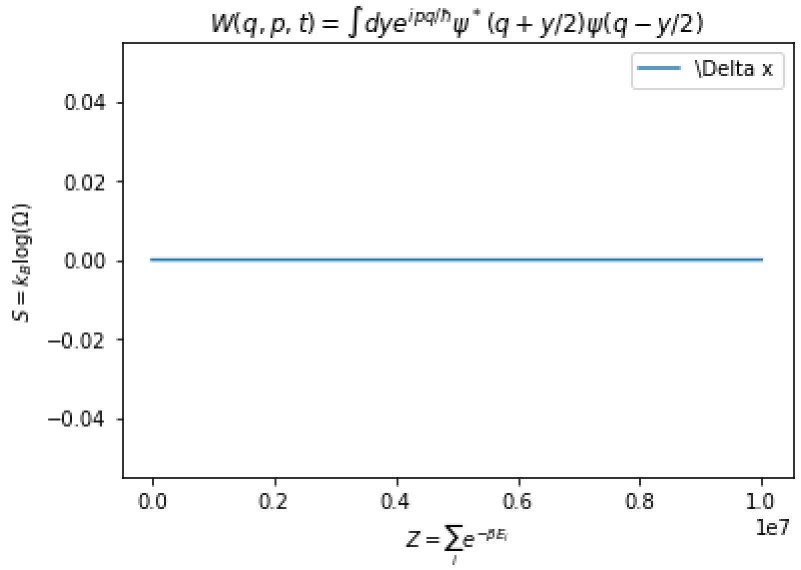
In [19]:
```python
plt.plot(Z-z,label='Difference')
plt.xlabel("Position")
plt.ylabel("$Z-z$")
plt.legend()
```

Out[19]:
```
<matplotlib.legend.Legend at 0x22c55599f28>
```



Look at the label on y. The font looks different. `MatPlotlib` supports $\LaTeX$!!!. Let us make the previous plot, but with hypotetical mathematical labels

```
In [20]: plt.plot(Z-z,label='\Delta x')
         plt.xlabel("$Z=\sum_i e^{- \\beta E_i}$")
         plt.ylabel("$S=k_B\log(\Omega)$")
         plt.title("$W(q,p,t)=\int dy e^{ipq/\hbar}\psi^*(q+y/2)\psi(q-y/2)$")
         plt.legend()

Out[20]: <matplotlib.legend.Legend at 0x22c1f096a20>
```



It can get even worse. Matrix sum.

First of all, let us see how can we construct a *matrix* using lists.

In [21]:
```
%%time
N=4
x=[list(range(i*N,N+i*N)) for i in range(N)]
```

```
Wall time: 167 ms
```

This is basically a list done of lists.

In [22]:
```
print(len(x),len(x[2]))
```

```
4 4
```

In [23]:
```
print(x)
```

```
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]]
```

Now, let us use a bigger matrices

In [24]:
```
%%time
N=10000
x=[list(range(i*N,N+i*N)) for i in range(N)]
y=[list(range(i*N,N+i*N)) for i in range(N)]
```

```
Wall time: 8.09 s
```

And summing them.

In [25]:
```
%%time
z=[]
for i in range(N):
    aux=[]
    for j in range(N):
        aux.append(x[i][j]+y[i][j])
    z.append(aux)
```

```
Wall time: 28.3 s
```

and just in the same way we did it for the case of *vectors*, we will convert the `list` s to `array` s

In [26]:
```
%%time
X=np.array(x)
Y=np.array(y)
```

```
Wall time: 20.5 s
```

And summing them,

In [27]:
```
%%time
Z=X+Y
```

```
Wall time: 952 ms
```

Look that in this case, we didn't have to define a new *sum*

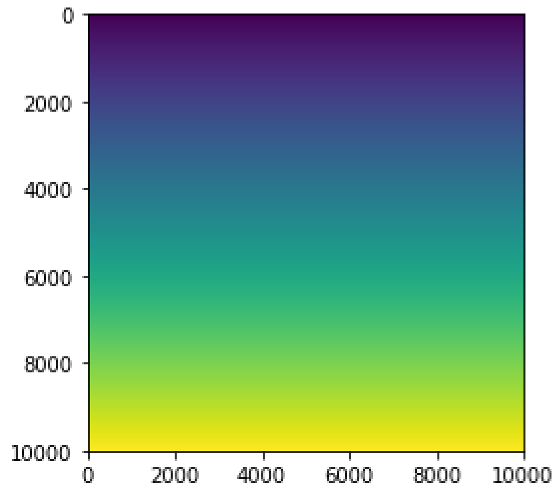
How is the difference on times?

> *HUGE!*

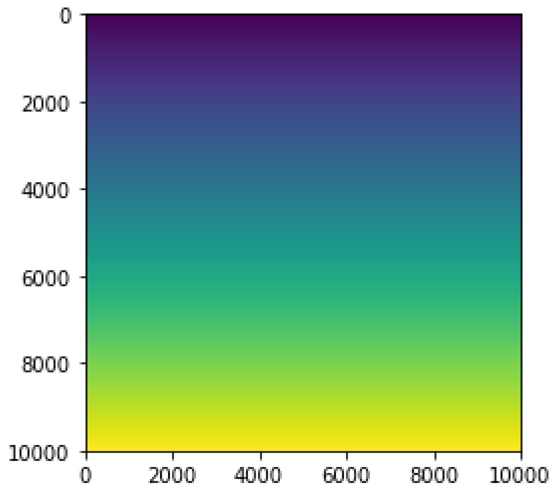
In the same sense than before, we can plot the matrices and their difference

In [28]:
```
plt.imshow(Z)
```

Out[28]:
```
<matplotlib.image.AxesImage at 0x22fd2801eb8>
```

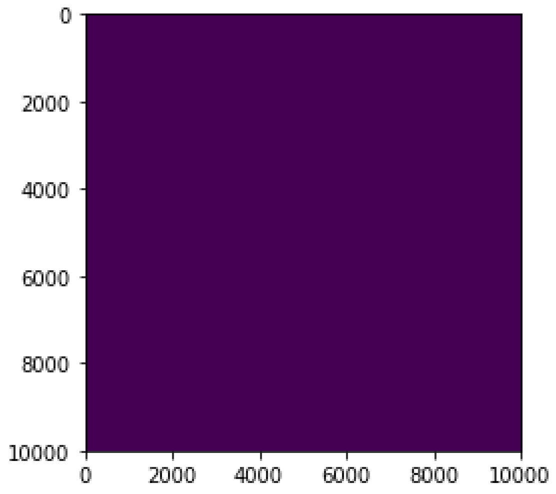


```
In [29]: plt.imshow(z)

Out[29]: <matplotlib.image.AxesImage at 0x22c1f1a8b70>
```



In [30]:
```
plt.imshow(Z-z)
```

Out[30]:
```
<matplotlib.image.AxesImage at 0x22c077bdb00>
```



But, what if we want to do a more rigurous study of the time certain process take?

We will use the library `time`

In [31]:
```
import time
```

The available methods are,

In [32]:
```
print(dir(time))
```

```
['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__name__', '__package__', '__spec__',
'altzone', 'asctime', 'clock', 'ctime', 'daylight', 'get_clock_info', 'gmtime', 'loca
ltime', 'mktime', 'monotonic', 'monotonic_ns', 'perf_counter', 'perf_counter_ns', 'pr
ocess_time', 'process_time_ns', 'sleep', 'strftime', 'strptime', 'struct_time', 'thre
ad_time', 'thread_time_ns', 'time', 'time_ns', 'timezone', 'tzname']
```

For instance,

```
In [33]: time.localtime()
```

```
Out[33]: time.struct_time(tm_year=2019, tm_mon=6, tm_mday=27, tm_hour=12, tm_min=54, tm_sec=2,
         tm_wday=3, tm_yday=178, tm_isdst=0)
```

We will concentrate on the function `time.time()`

This function counts how many seconds have passed since 1 Jan. 1970.

```
In [34]: time.time()

Out[34]: 1561658042.7488396
```

Without being rigorous, one can more or less test that,

```
In [35]: time.time()/60/60/24/365

Out[35]: 49.5198516864145

In [36]: 2019-1970

Out[36]: 49
```

So, we will measure the time elapsen on a given command as,

```
In [37]: a=time.time()
         time.sleep(2)
         b=time.time()
```

So that the difference is now the amount of seconds,

```
In [38]: b-a

Out[38]: 2.0002174377441406
```

For instance, on our example of the time it takes to sum two built in `list` s

In [39]:
```
a=time.time()
N=10000000
x=list(range(N))
y=list(range(N))
z=[]
b=time.time()
for i in range(len(x)):
    z.append(x[i]+y[i])
c=time.time()
print(c-b,"sec",c-a,"sec")
```

```
2.331831455230713 sec 12.639823913574219 sec
```

We would like to have *something* that, given the size returns the time it takes, so that we can plot the time consumed by the computer for a sum of a given size pair of vectors.

Functions

A function is a peace of code, that can be *called* without typing the comands again, the structure goes as,

```
In [40]: def test1(a,b):
             return a*b
```

```
In [41]: test1(2,3)

Out[41]: 6

In [42]: test1(2.0,3.0)

Out[42]: 6.0

In [43]: test1([1,2],3)

Out[43]: [1, 2, 1, 2, 1, 2]

In [44]: test1("a",10)

Out[44]: 'aaaaaaaaaa'
```

Our function should

- Receive the size N
- Generate the vectors $\mathbf{x}$ and $\mathbf{y}$
- Create the Output vector $\mathbf{z}$
- Sum $\mathbf{z} = \mathbf{x} + \mathbf{y}$

In [45]:

```
def list_sum(N):
    x=list(range(N))
    y=list(range(N))
    z=[]
    b=time.time()
    for i in range(len(x)):
        z.append(x[i]+y[i])
    c=time.time()
    return c-b
```

Let us test it,

```
In [46]: list_sum(1)
Out[46]: 0.0

In [47]: list_sum(10)
Out[47]: 0.0
```

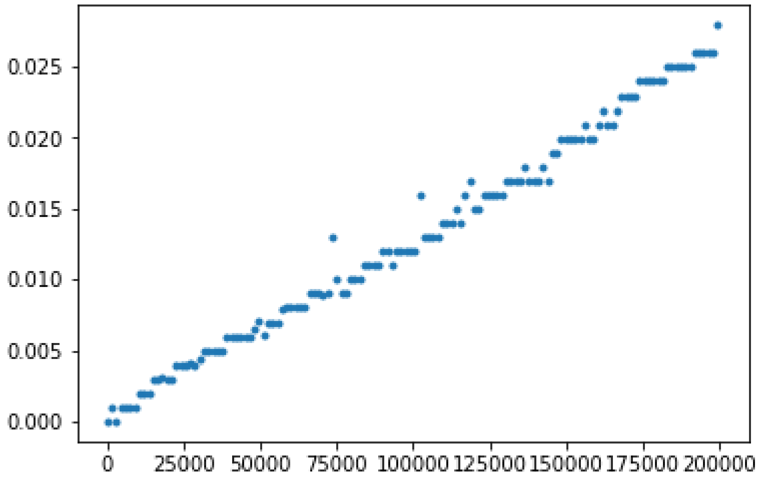
```
In [48]: list_sum(100)

Out[48]: 0.0006272792816162109

In [49]: list_sum(1000)

Out[49]: 0.0
```

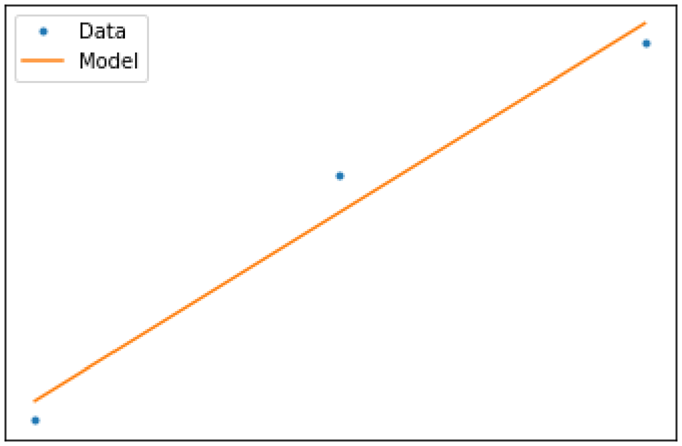
```
In [50]: list_sum(10000)

Out[50]: 0.0020236968994140625

In [51]: list_sum(100000)

Out[51]: 0.013962507247924805

In [52]: list_sum(10000000)

Out[52]: 1.506976842880249
```

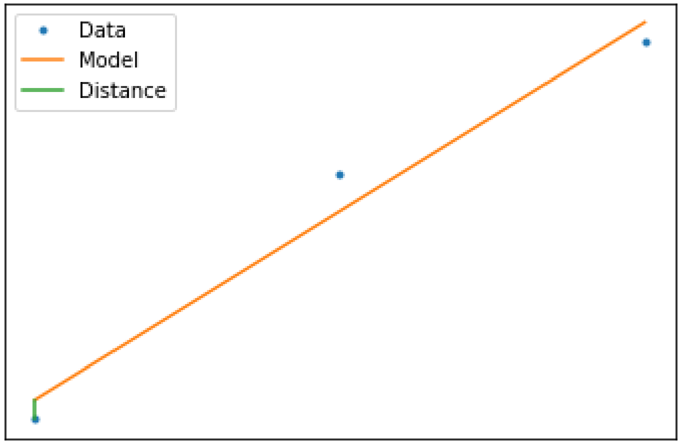
In [53]:
```python
times=[]
t=[]
for i in range(0,200000,1500):
    t.append(i)
    times.append(list_sum(i))
t=np.array(t)
```

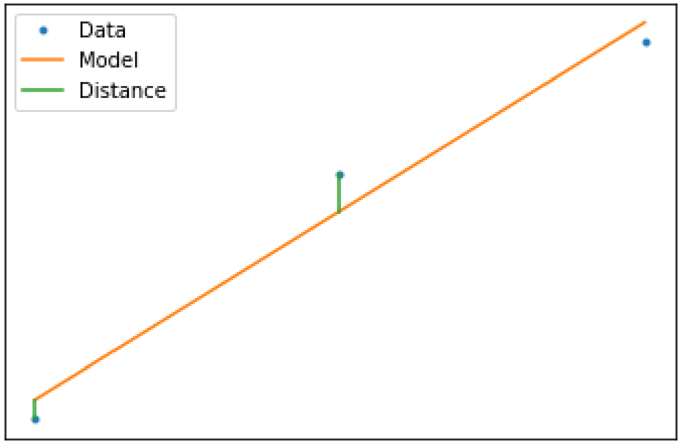
```
In [54]: plt.plot(t,times,'.')
```

```
Out[54]: [<matplotlib.lines.Line2D at 0x22c30aa4630>]
```

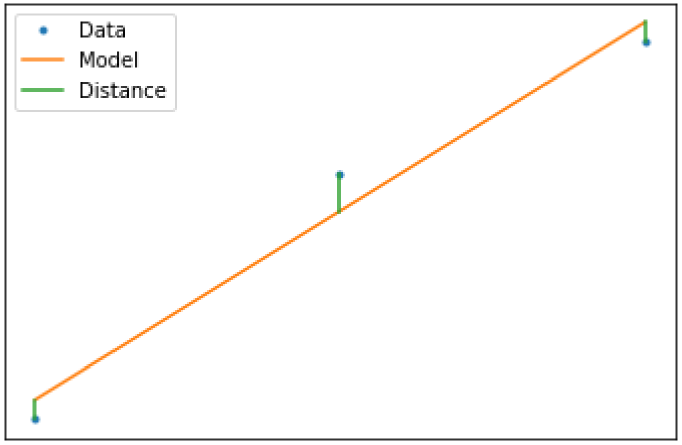



Data
Model


Data
Model
Distance



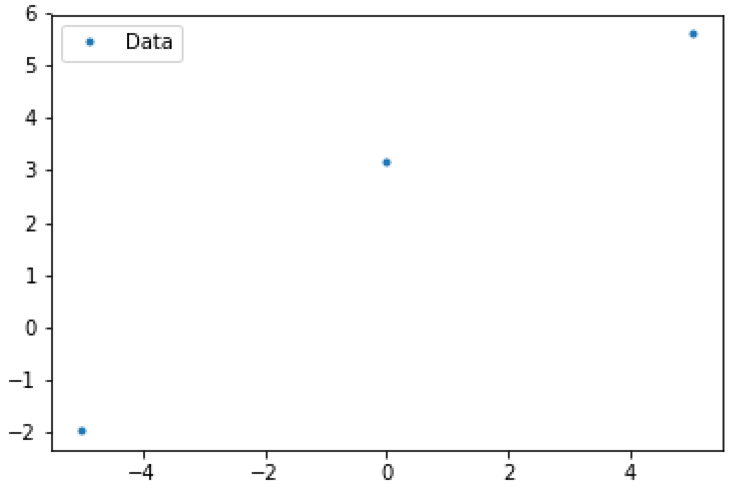
Data
Model
Distance

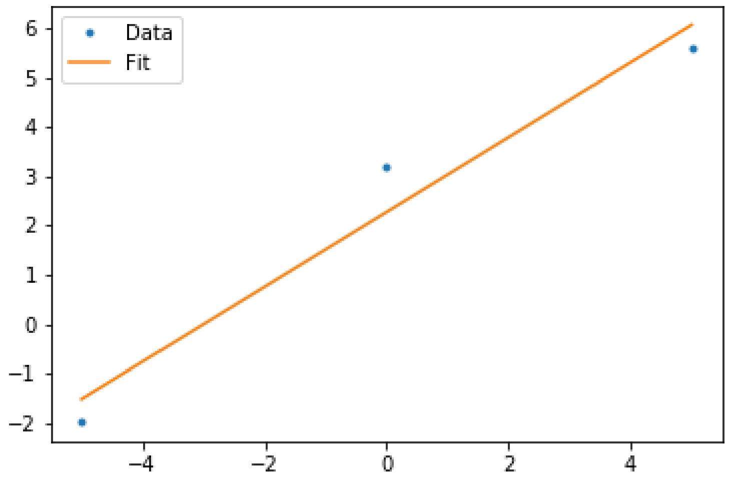

Data
Model
Distance

That means that

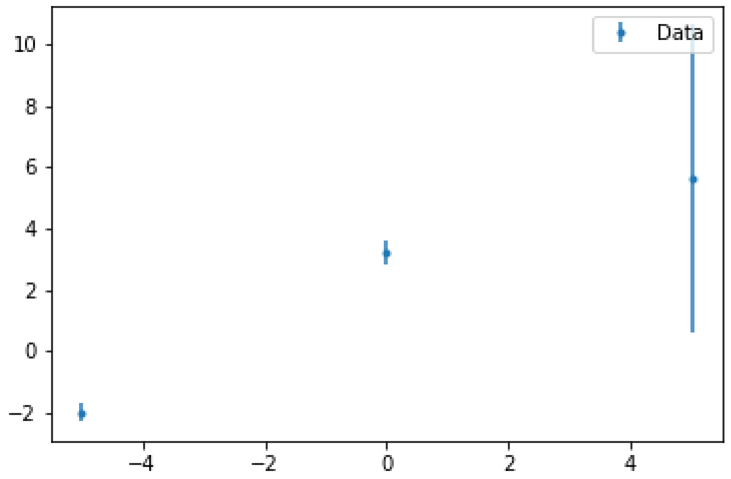
$$d_{\text{Green lines}} = \sum_i d_i(y - f(x))^2$$

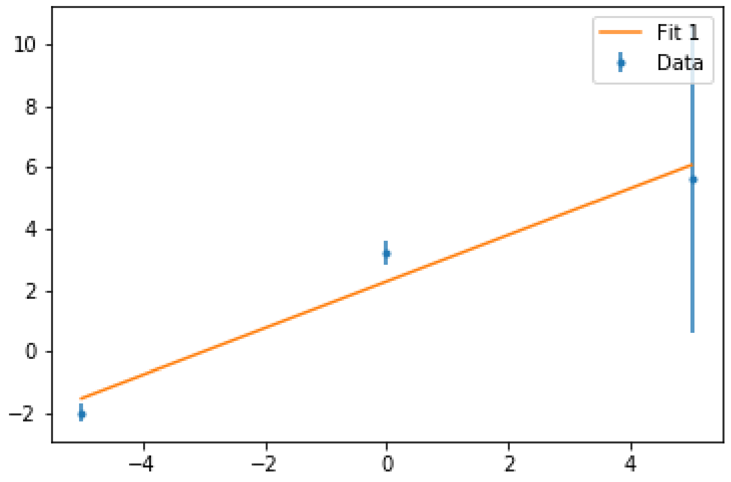
such that $d_{\text{Green lines}}$ takes its minumun value.


Data
6
5
4
3
2
1
0
−1
−2
−4
−2
0
2
4

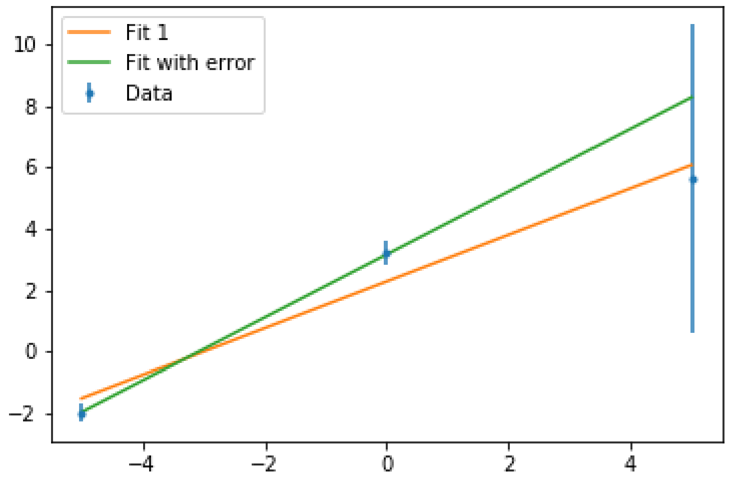


Data
Fit
6
5
4
3
2
1
0
−1
−2
−4
−2
0
2
4


Data
10
8
6
4
2
0
−2
−4
−2
0
2
4



Fit 1
Data
10
8
6
4
2
0
−2
−4
−2
0
2
4



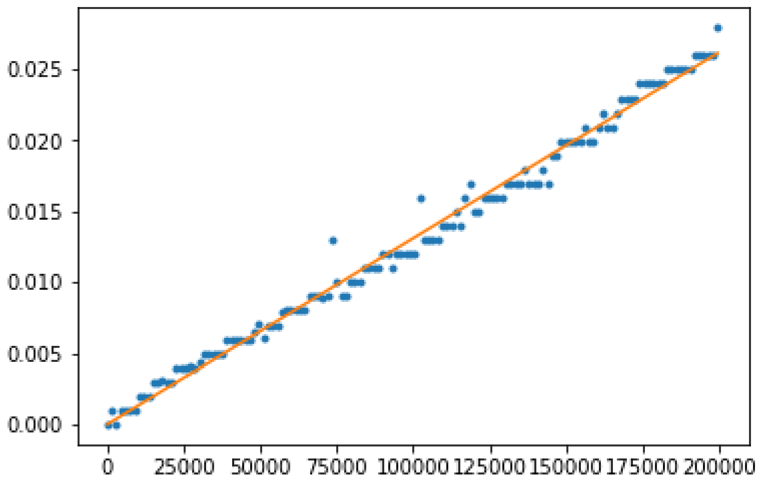
Fit 1
Fit with error
Data
10
8
6
4
2
0
−2
−4
−2
0
2
4

```
In [55]: from scipy.optimize import curve_fit
```

In [56]:
```
def f(x,a):
    return a*x
```

In [57]:
```
popt,pcov=curve_fit(f,t,times)
```

```
In [58]: plt.plot(t,times,'.')
         plt.plot(t,f(np.array(t),*popt))

Out[58]: [<matplotlib.lines.Line2D at 0x22c31123588>]
```



```
In [59]: print(pcov)
```

```
[[3.19685454e-19]]
```

```
In [60]: print(popt)
```

```
[1.30912978e-07]
```

```
In [61]: print(np.sqrt(pcov.diagonal()))
```

```
[5.65407335e-10]
```

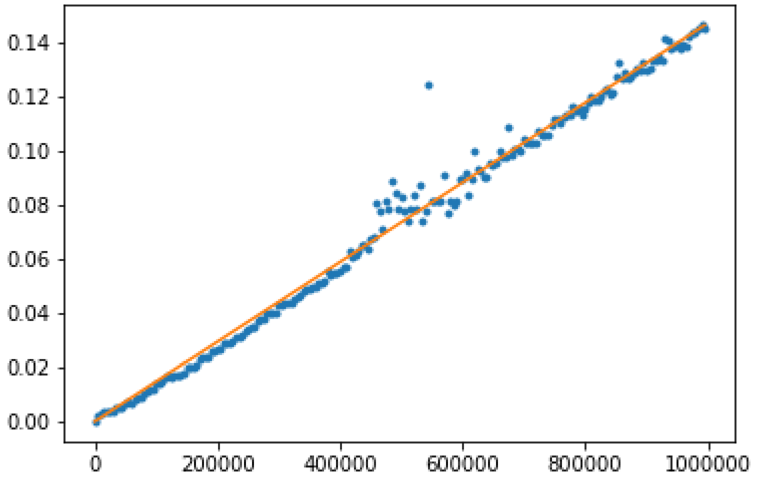
In [62]:

```
times=[]
t1=[]
for i in range(0,1000000,5000):
    t1.append(i)
    times.append(list_sum(i))
t1=np.array(t1)
```

In [63]:
```
popt,pcov=curve_fit(f,t1,times)
```

In [64]:
```
plt.plot(t1,times,'.')
plt.plot(t1,f(t1,*popt))
```

Out[64]:
```
[<matplotlib.lines.Line2D at 0x22c31158048>]
```



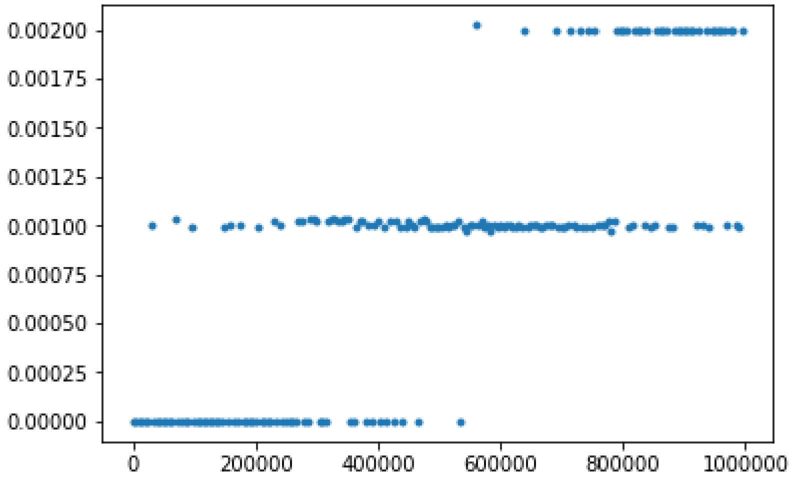
In [65]:

```
def arr_sum(N):
    x=np.arange(N)
    y=np.arange(N)
    b=time.time()
    z=x+y
    c=time.time()
    return c-b
```

In [66]:

```
times=[]
t2=[]
for i in range(0,1000000,5000):
    t2.append(i)
    times.append(arr_sum(i))
t2=np.array(t2)
```

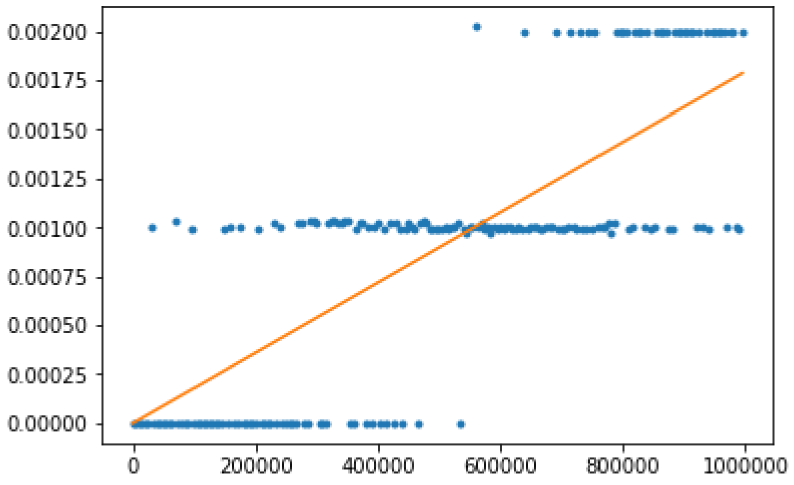
```
In [67]: plt.plot(t2,times,'.')

Out[67]: [<matplotlib.lines.Line2D at 0x22c31247fd0>]
```



```
In [68]: popt2,pcov2=curve_fit(f,t2,times)
```

```
In [69]: plt.plot(t2,times,'.')
         plt.plot(t2,f(t2,*popt2))
```

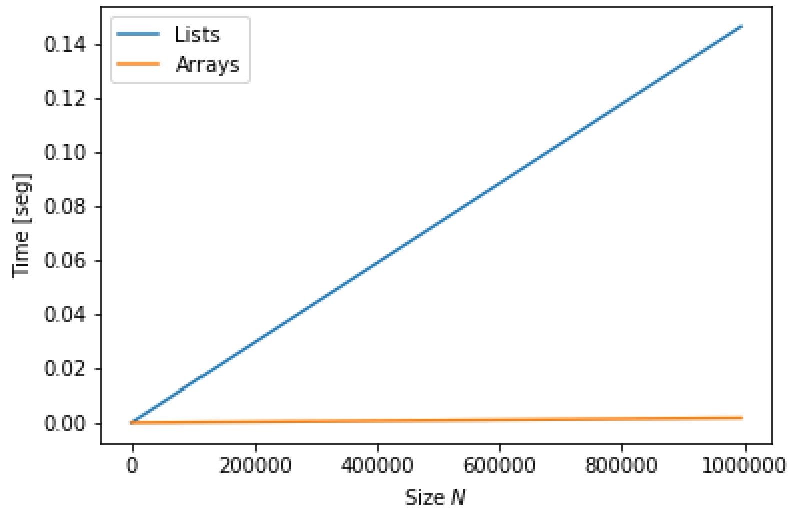
```
Out[69]: [<matplotlib.lines.Line2D at 0x22c31283978>]
```



```
In [70]: print(pcov2.diagonal()**.5)
```

```
[5.18637054e-11]
```

```
In [71]: plt.xlabel("Size $N$")
         plt.ylabel("Time [seg]")
         plt.plot(t1,f(t1,*popt),label='Lists')
         plt.plot(t2,f(t2,*popt2),label='Arrays')
         plt.legend()

Out[71]: <matplotlib.legend.Legend at 0x22c31315dd8>
```



```
In [72]: print(popt,popt2)
```

```
[1.47226817e-07] [1.79354508e-09]
```