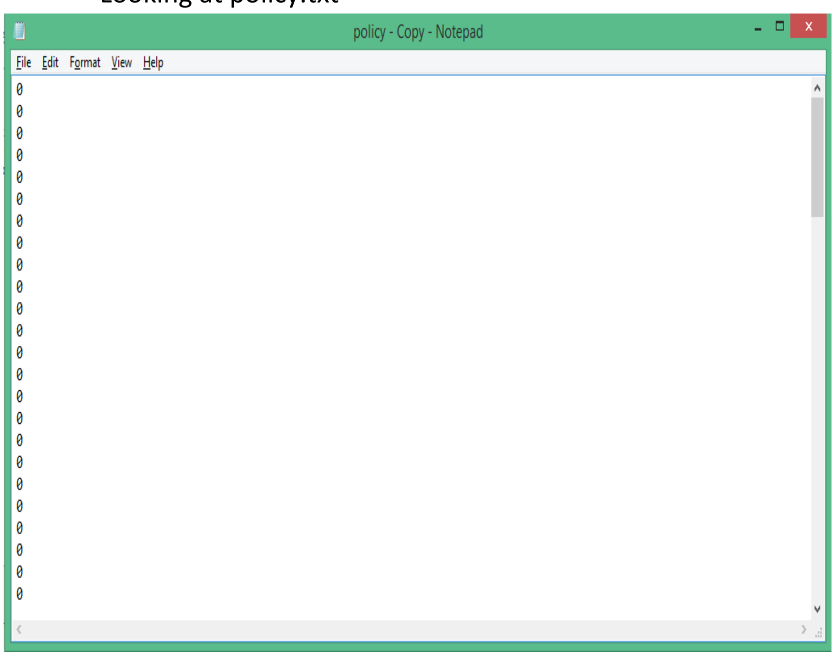
1) (a)

It is easy to observe that with $\epsilon = 0$, The system get some negative rewards at the beginning and fast learn a policy that gives zero.

Looking at policy.txt



average:	-0.00095
stdev:	0.053142

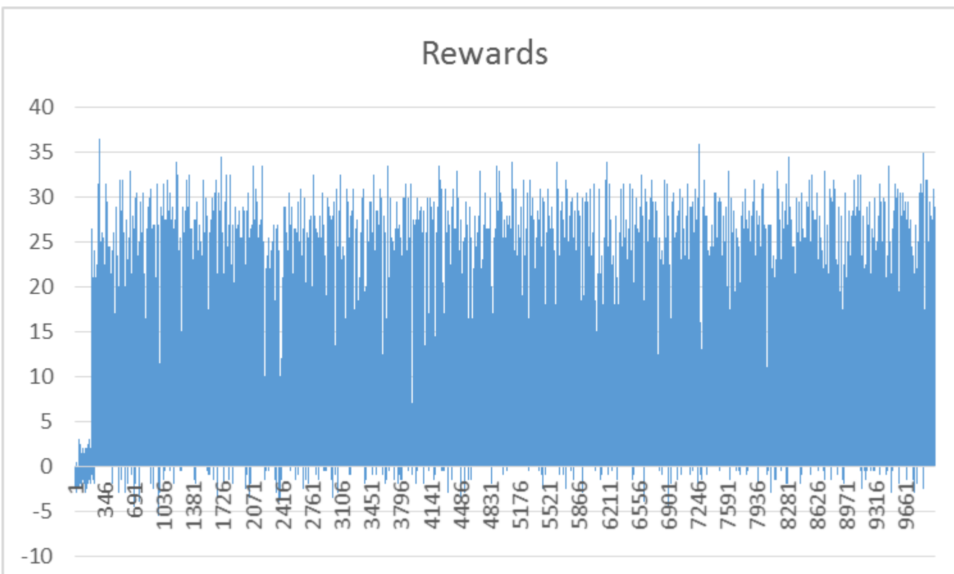
Almost all the other elements are zero. This means that the policy that the system learning is to move just in the left direction.

		#		1
		#		
		#	#	#
		#		
C		#		2

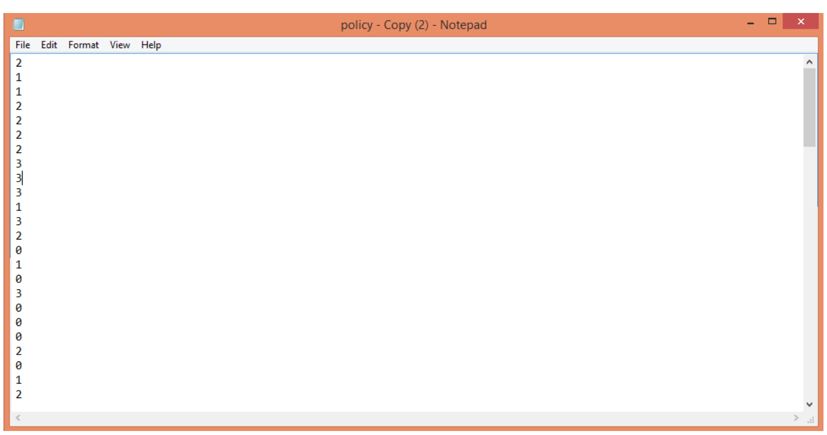
These make sense, since with we look the third column, it has just slipperiness states. So, without any random decision, the robot learned that it should stay on the left side, because these way it will not get negative rewards by the slipperiness.

(b)

In the example with $\varepsilon = 0.1$, during the first episodes, we can identify very low rewards, including negatives ones. However, because of the randomness, the robot learned that it can pass throw the third column and gain positive high rewards delivering.



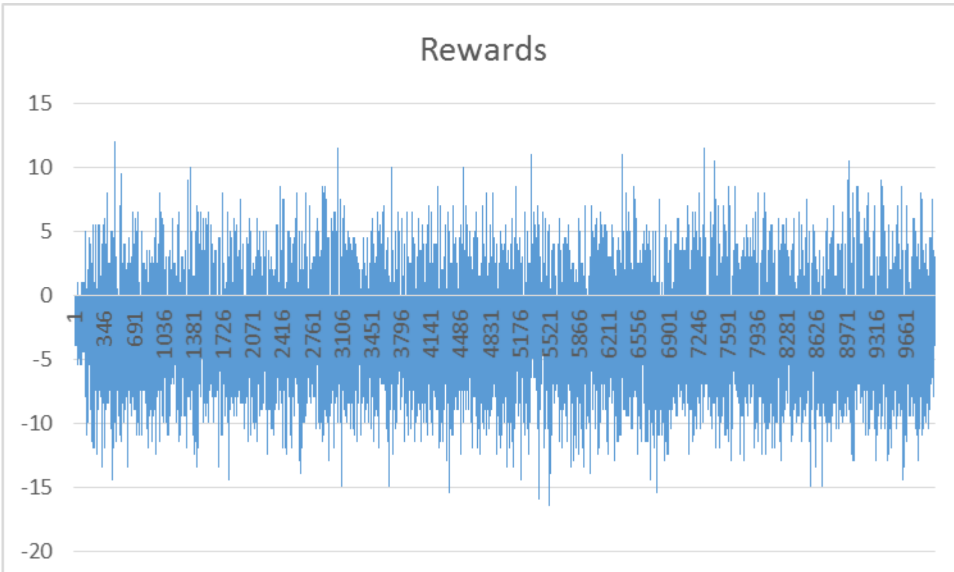
Looking policy, we see a complete different behavior between (a) and (b).



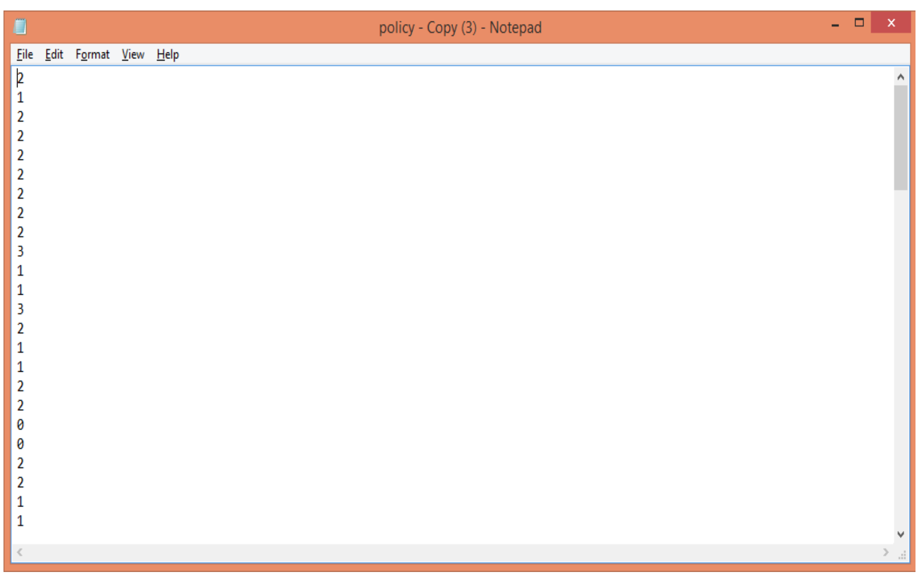
average:	13.60105
stdev:	8.683258

(c) Setting up $\varepsilon = 0.5$, We give too much importance to the randomness. It cause problems, since instead of looking for the utility, which gives the believed best direction, it goes often to a random direction, which leads to the thief and to slipperiness positions.

We should point out that rewards assume positive and negative values. It could be explained that the randomness sometimes will help other mess the system, causing these stochastic behavior.



Policies result are shown below.



policy - Copy (3) - Notepad

File Edit Format View Help

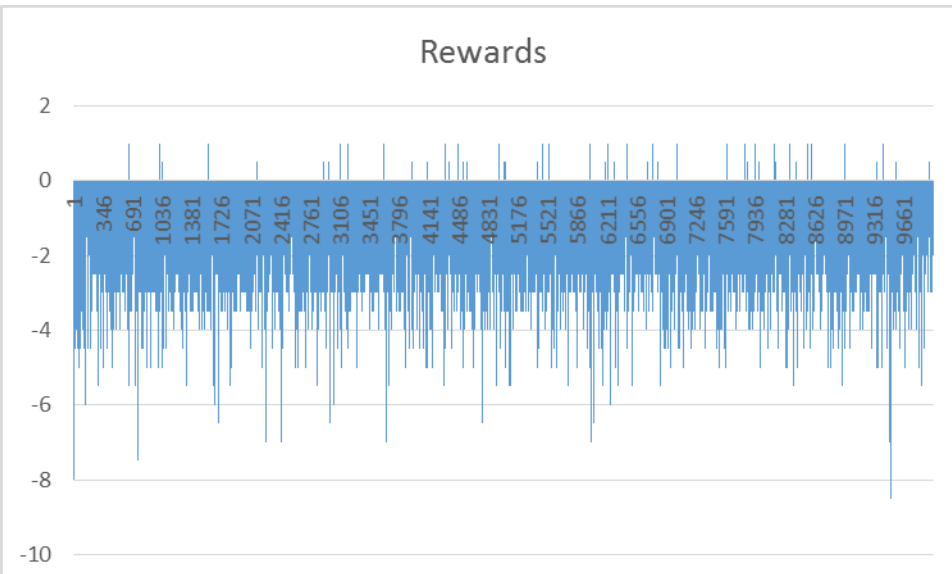
```
2
1
2
2
2
2
2
2
2
3
1
1
3
2
1
1
2
2
0
0
2
2
1
1
```

average:	-2.97805
stdev:	3.932492

2)

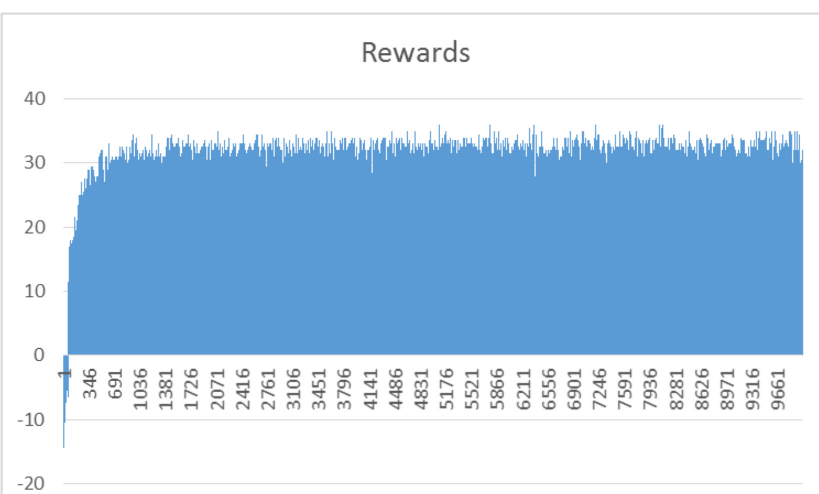
	#	T		1
	#	T	#	#
		T		
	#	T		
C	#	T		2

(a) Since the robot has no idea where is the thief, and for sure he is in the third column. Every time the robot passes in the third column, there is a probability $1/5$ to meet the thief. We should point out too that lossByThief is extremely high (Equal the sum of two successful delivery). Because of these explanation, be in worldWithThief without knowing where he is, reduce the rewards.



average:	-1.30725
stdev:	1.025584

(b) Know where the thief is, make the robot avoid collisions with him. That said, the reward will increase a lot because losses by thief will reduce drastically, and the penalty lossByThief is extremely high compared to other. As shown below, the average increases a lot.



average:	27.2753
stdev:	4.503911

(c) To achieve the best ε. I changed the given classes of these MP. Create to method in the interface Agent

```
public void setEpsilon(double value);

public double getEpsilon();
```

I implemented both of them, and create a loop in simulator class, which will simulate for some ε.

```
for (double i = 0.0; i < 0.5 ; i+= 0.01) {

        agent.setEpsilon(i);

        (new Simulator(world, agent, thiefKnown, steps, episodes, policyOutput,
episodeOutput)).simulate();

}
```

Finally I calculate the average and print it.

```
double sum = 0;

for(Double value : episodeList){

         sum += value;

}

System.out.println("epsilon: " + this.agent.getEpsilon() + "  value: " + (sum/this.episodes));
```

Simulating for these boundaries, the results are shown below.

```
epsilon: 0.0  value: 13.9256
epsilon: 0.01  value: 30.95795
epsilon: 0.02  value: 30.9464
epsilon: 0.03  value: 29.41485
epsilon: 0.04  value: 28.3305
epsilon: 0.05  value: 27.26705
epsilon: 0.060000000000000005  value: 25.92235
epsilon: 0.07  value: 24.63915
epsilon: 0.08  value: 23.3991
epsilon: 0.09  value: 22.0207
epsilon: 0.09999999999999999  value: 20.9608
epsilon: 0.10999999999999999  value: 19.48365
epsilon: 0.11999999999999998  value: 18.343
epsilon: 0.12999999999999998  value: 17.2957
epsilon: 0.13999999999999999  value: 16.08305
epsilon: 0.15  value: 14.96195
epsilon: 0.16  value: 13.5857
epsilon: 0.17  value: 12.4268
epsilon: 0.18000000000000002  value: 11.22895
epsilon: 0.19000000000000003  value: 10.1751
epsilon: 0.20000000000000004  value: 8.92255
epsilon: 0.21000000000000005  value: 7.77845
epsilon: 0.22000000000000006  value: 6.74865
epsilon: 0.23000000000000007  value: 5.7501
epsilon: 0.24000000000000007  value: 4.6529
epsilon: 0.25000000000000006  value: 3.38555
epsilon: 0.26000000000000006  value: 2.4789
epsilon: 0.2700000000000001  value: 1.2507
epsilon: 0.2800000000000001  value: 0.28085
epsilon: 0.2900000000000001  value: -0.90185
epsilon: 0.3000000000000001  value: -1.80565
epsilon: 0.3100000000000001  value: -2.6328
epsilon: 0.3200000000000001  value: -3.6839
epsilon: 0.3300000000000001  value: -4.7738
epsilon: 0.34000000000000014  value: -5.6704
epsilon: 0.35000000000000014  value: -6.60825
epsilon: 0.36000000000000015  value: -7.457
epsilon: 0.37000000000000016  value: -8.28375
epsilon: 0.38000000000000017  value: -9.13295
epsilon: 0.3900000000000002  value: -10.0451
epsilon: 0.4000000000000002  value: -10.8722
epsilon: 0.4100000000000002  value: -11.80535
epsilon: 0.4200000000000002  value: -12.5021
epsilon: 0.4300000000000002  value: -13.11275
epsilon: 0.4400000000000002  value: -13.8996
epsilon: 0.45000000000000023  value: -14.6594
epsilon: 0.46000000000000024  value: -15.4155
epsilon: 0.47000000000000025  value: -15.98525
epsilon: 0.48000000000000026  value: -16.51895
epsilon: 0.49000000000000027  value: -17.26625
```

Simulating again for ε between 0 and 0.03.

```
epsilon: 0.0  value: 14.0801
epsilon: 0.001  value: 18.45155
epsilon: 0.002  value: 26.85375
epsilon: 0.003  value: 27.5455
epsilon: 0.004  value: 29.49235
epsilon: 0.005  value: 29.7079
epsilon: 0.006  value: 30.4492
epsilon: 0.007  value: 30.1329
epsilon: 0.008  value: 30.41375
epsilon: 0.009000000000000001  value: 30.63905
epsilon: 0.010000000000000002  value: 30.97965
epsilon: 0.011000000000000003  value: 30.6944
epsilon: 0.012000000000000004  value: 30.9462
epsilon: 0.013000000000000005  value: 31.38215
epsilon: 0.014000000000000005  value: 30.75225
epsilon: 0.015000000000000006  value: 31.0765
epsilon: 0.016000000000000007  value: 30.8907
epsilon: 0.017000000000000008  value: 30.54255
epsilon: 0.01800000000000001  value: 30.93385
epsilon: 0.01900000000000001  value: 29.7535
epsilon: 0.02000000000000001  value: 30.7779
epsilon: 0.02100000000000001  value: 30.6344
epsilon: 0.022000000000000013  value: 30.31125
epsilon: 0.023000000000000013  value: 30.4204
epsilon: 0.024000000000000014  value: 30.3183
epsilon: 0.025000000000000015  value: 30.1591
epsilon: 0.026000000000000016  value: 30.04555
epsilon: 0.027000000000000017  value: 29.3791
epsilon: 0.028000000000000018  value: 29.78165
epsilon: 0.02900000000000002  value: 29.16195
```

After seven simulations, was detected a pattern that the best ε stays between 0.1 and 0.16. And in four out of these seven cases $\varepsilon = 0.013$ got the highest average. So I choose 0.013 as the best ε

Analogously, we use the same method to obtain the learning rate. As firsts results, we got.

```
learning rate: 0.0  value: -11.03735
learning rate: 0.1  value: 31.2188
learning rate: 0.2  value: 31.637
learning rate: 0.30000000000000004  value: 29.996
learning rate: 0.4  value: 30.10915
learning rate: 0.5  value: 29.1952
```

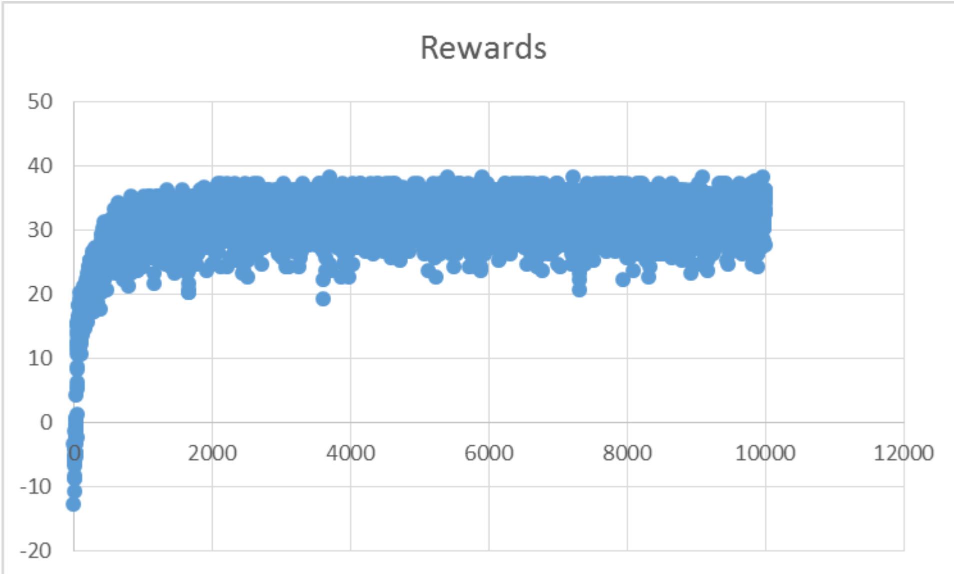
Assuming that the best learning rate stays between 0.1 and 0.3. We change the boundaries for the learning rate loop. The results are shown below.

```
learning rate: 0.1  value: 30.96345
learning rate: 0.11  value: 30.93685
learning rate: 0.12  value: 31.2486
learning rate: 0.13  value: 30.8777
learning rate: 0.14  value: 31.31625
learning rate: 0.15000000000000002  value: 30.57005
learning rate: 0.16000000000000003  value: 31.34865
learning rate: 0.17000000000000004  value: 31.67225
learning rate: 0.18000000000000005  value: 30.96145
learning rate: 0.19000000000000006  value: 30.8895
learning rate: 0.20000000000000007  value: 31.37855
learning rate: 0.21000000000000008  value: 31.4945
learning rate: 0.22000000000000008  value: 31.35025
learning rate: 0.2300000000000001  value: 31.36115
learning rate: 0.2400000000000001  value: 30.9815
learning rate: 0.2500000000000001  value: 30.21405
learning rate: 0.2600000000000001  value: 30.9473
learning rate: 0.27000000000000013  value: 31.0652
learning rate: 0.28000000000000014  value: 31.0638
learning rate: 0.29000000000000015  value: 30.94895
```

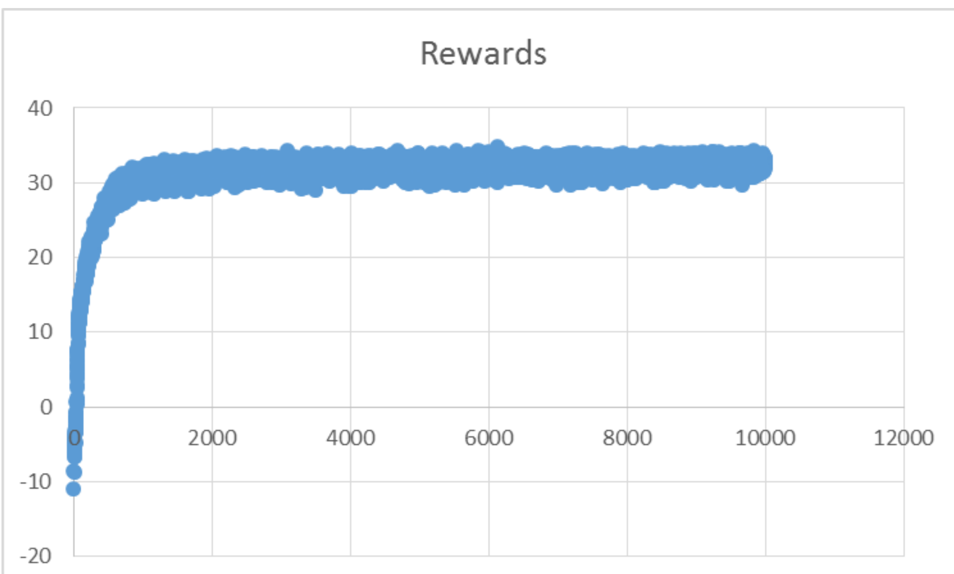
Seven simulations were done. Based in all of them, we can conclude that α stays between 0.16 and 0.24. We choose $\alpha = 0.20$ as best value.

3)

Setting up all α and ε. The first simulation is shown below.



We made nine more simulations. The result of the average of them is shown below.



The First graphics has samples more spread over the y-axis, while the average graphics, all samples are more close to concentrate in a trendline that seems to be exponential.

We could associate a random factor over the calculation of each total rewards. So

$$totalRewards = a + e$$

Where a follows a trendline and e is a random variable. When we print the first simulation, e make the samples spread over the y axis.

For the second case, that is the average, we will print something similar too

$$totalRewards = \frac{a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + a_9 + a_{10}}{10} + \frac{e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7 + e_8 + e_9 + e_{10}}{10}$$

$$but\ a_i = a$$

$$totalRewards = a + \frac{e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7 + e_8 + e_9 + e_{10}}{10}$$

So, the difference is that the random factor is now a combination linear of random variables. It is known that.

$$stdev\ (\alpha\, x + \beta\, y) = \sqrt{\alpha^2 stdev(x) + \beta^2 stdev(y)}$$

These result will make the standard deviation of $\frac{e_1+e_2+e_3+e_4+e_5+e_6+e_7+e_8+e_9+e_{10}}{10}$ reduce face the standard deviation of e_1, because the sum of square root of squares is less them the average which is a good estimation for stdev of e_1.

Thinking less mathematically, what occurs is that when a random variable assumes a high value, the other ones will compensate these value, and there is a $1/10$ factor to attenuate. All of these reduces the standard deviation. And Because of these reduction, the second graphs has samples closer to each other when compared to the first Graph.