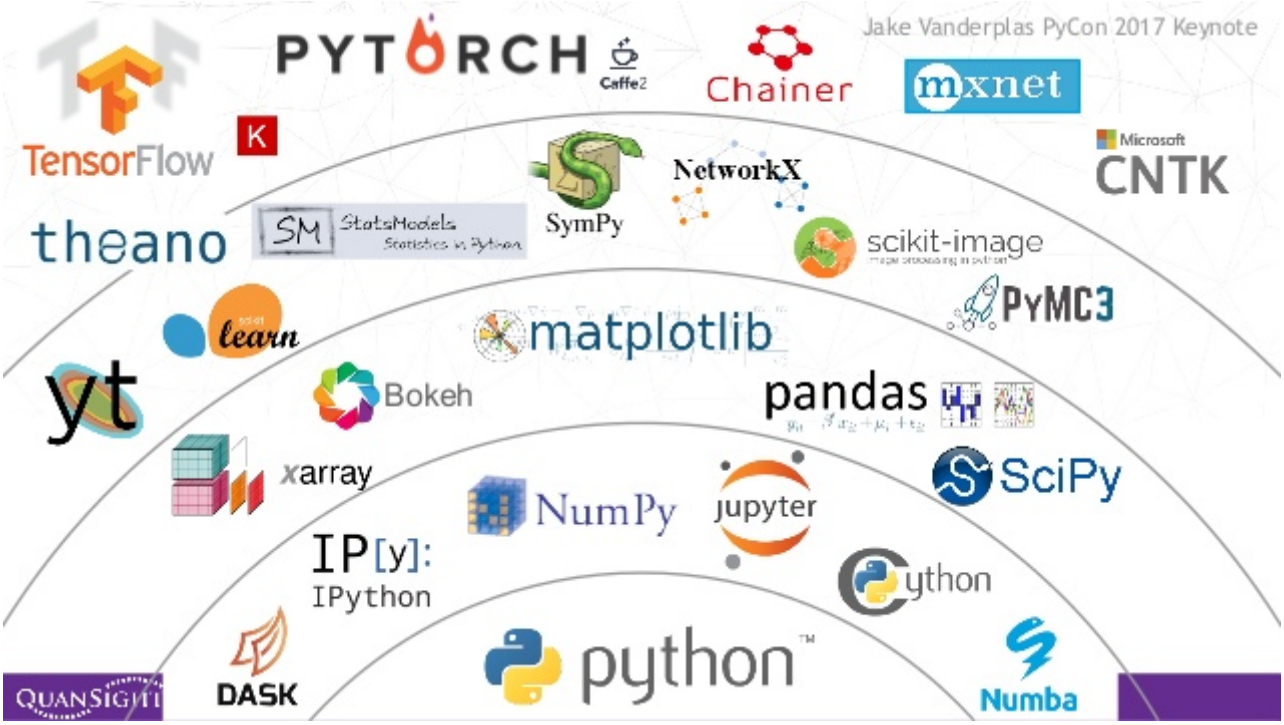
First Practice ML

Python has a lot of tools and libraries for machine learning, data science, mathematics, and science. Some of them:



In this practice, we will get to know some important python data science libraries in the **SciPy** Ecosystem, that every data scientist must be familiar with.
We will also learn about the **MSE** loss function.

SciPy Ecosystem

Link: https://www.scipy.org/ (https://www.scipy.org/)

SciPy (pronounced “Sigh Pie”) is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:

NumPy

Link: https://numpy.org/ (https://numpy.org/)

NumPy is a fundamental package for scientific computing with Python. The library makes the use of N-dimensional arrays easy and user-friendly. It is the base library for all the SciPy ecosystem.

Pandas

Link: https://pandas.pydata.org/ (https://pandas.pydata.org/)

Pandas is a fast, powerful, flexible, and easy to use open-source data analysis and manipulation tool, built on top of the Python programming language.

Matplotlib

Link: https://matplotlib.org/ (https://matplotlib.org/)

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

IPython

Link: http://ipython.org/ (http://ipython.org/)

The IPython project provides an enhanced interactive environment that includes, among other features, support for data visualization and facilities for distributed and parallel computation.
Most used with Jupyter Notebook.

Jupyter Notebook

Link: https://jupyter.org/ (https://jupyter.org/)

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. Uses include data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

SciPy Library

Link: https://www.scipy.org/scipylib/index.html (https://www.scipy.org/scipylib/index.html)

The SciPy library is one of the core packages that make up the SciPy stack. It provides many user-friendly and efficient numerical routines, such as routines for numerical integration, interpolation, optimization, linear algebra, and statistics.

SymPy

Link: [https://www.sympy.org/en/index.html (https://www.sympy.org/en/index.html)](https://www.sympy.org/en/index.html)

SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible to be comprehensible and easily extensible.

Imports

Most of the time, we'll want to put all of our imports and initializations in one place at the start of the notebook.
This is particularly true when we want to download and install things on the kernel (the Linux computer that runs the notebook).
Here I will put `import` s right before their use, so you can see when they are necessary.
We can also open a terminal in a cell.
For this to work, we need to install `kora` . We do it with the `pip install` command.
To run shell commands in cells (without a terminal) we need to put `!` at the start of the row.
To activate a cell, just click on it, and press the `shift` + `enter` buttons.

In [ ]:

```
# install the `kora` library with `pip install`
!pip install opencv-python
```

```
Requirement already satisfied: opencv-python in /usr/local/lib/python3.7/d
ist-packages (4.1.2.30)
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/d
ist-packages (from opencv-python) (1.19.5)
```

NumPy

We use NumPy to create and manipulate a list of numbers.
We want to create points and show them in a graph.
These are the x values of our points.

In [ ]:

```
# import numpy and create array of 100 numbers from 0 to 99
import numpy as np
x = np.arange(100)
x
```

Out[ ]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

To create the y values of the points, we need to decide on a function.
Let's use the indentity function `y(x) = x` .

In [ ]:

```
y = x.copy()
y
```

Out[ ]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

Matplotlib

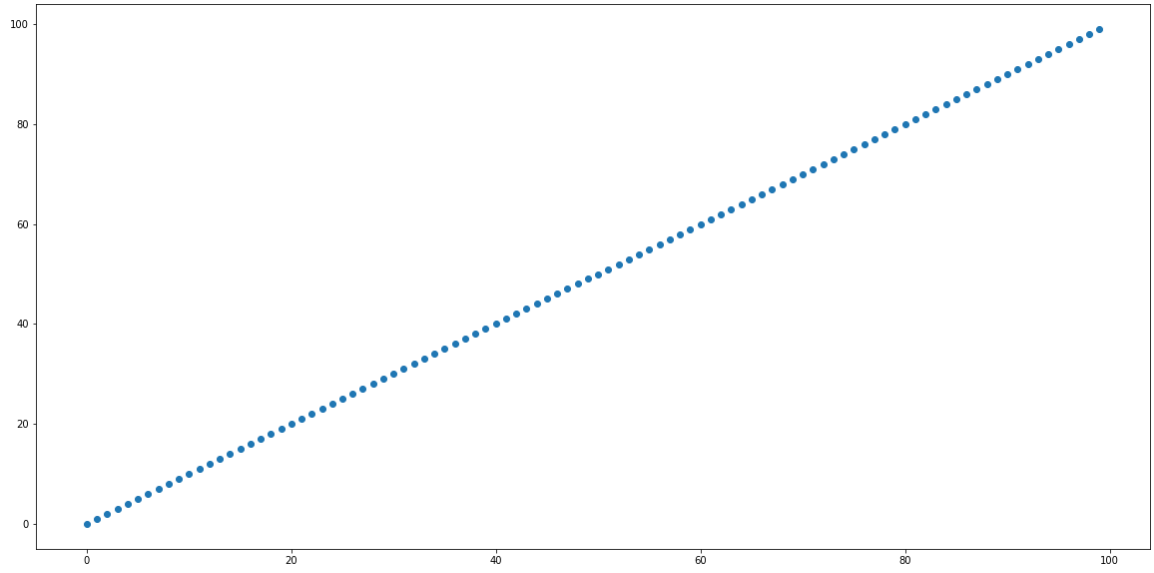
We use Matplotlib.pyplot to show graphs.
We can plot the dots in a scatter plot, to show the function with the dots.

In [ ]:

```
# import matplotlib.pyplot, enlarge figure size and create scatter plot
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (20,10)
plt.scatter(x, y)
```

Out[ ]:

```
<matplotlib.collections.PathCollection at 0x7efd035db860>
```



We created and showed the function `y(x) = x'.
Let's try to add some noise (also called error) to the function.
We do it with random noise in normal distribution (https://en.wikipedia.org/wiki/Normal_distribution), with standard deviation (https://en.wikipedia.org/wiki/Standard_deviation) of 5.

In [ ]:

```
# create noise with normal distribution with mean=0 and standard_deviation=5
eps = np.random.normal(loc=0, scale=5, size=[len(y)])
eps
```

Out[ ]:

```
array([  3.03517893,  -3.82117412,   3.63641171,  -1.77281532,
         7.53161315,   1.85036967,   1.02046492,   0.51948556,
        -0.09919262,  -6.69721499,   0.46915097, -10.53898066,
        -7.24471183,  -8.86457753,  -3.39646391,  -5.68137419,
         4.68257291,  -3.20539443,  -5.89444045,  -9.87528936,
         5.55078676,  -1.06340451,  -0.45744355,   1.59587454,
        -3.642191  ,   1.15390175,  -1.27947054,  -6.15180672,
         1.67710055,  -4.61461344,  -5.06431238,   7.01629247,
        -1.73229794,   2.54897579,   2.60704533,   2.32712445,
        -2.0834272 ,   1.6949391 ,   6.13655623,   1.04854594,
         2.38126495,   9.44758534,  -4.729907  ,   3.09763445,
        -3.91969174,  -1.76207054,   3.97387177,   6.84766767,
         2.97640145,   1.30539833,   2.59859053,  -0.42230329,
         6.88362419,  -7.56904211,  -3.29705065,   1.08892189,
         2.84146136,  -2.60221376,   1.9306173 ,   0.01749713,
         0.88173735,  -0.95197163,   4.15927616,  -6.00826158,
         6.72263381,   2.94630284, -12.83792571,   3.74283331,
        -2.10245848,  -4.23556737,  -0.14743304,  -7.41140429,
        10.212065  ,   1.87262769,  -2.24875998,  -4.20768576,
        -4.67117438,   1.01681038,  -0.40427497, -11.24713179,
        -2.02571397,  -2.10244059,  -3.67660881,  -1.4572848 ,
        -5.36369546,   0.96496243,  -2.0608571 ,   0.96975109,
        -5.5332064 ,   1.73401437,   3.19535762,  -5.65565404,
        -1.73001274,  -0.505855  ,   1.22880342,   7.08225711,
         1.17475512,   0.19108117,  -0.5606122 ,  -6.82838614])
```

In [ ]:

```
# add the noise to the function points y values
new_y = y + eps
new_y
```

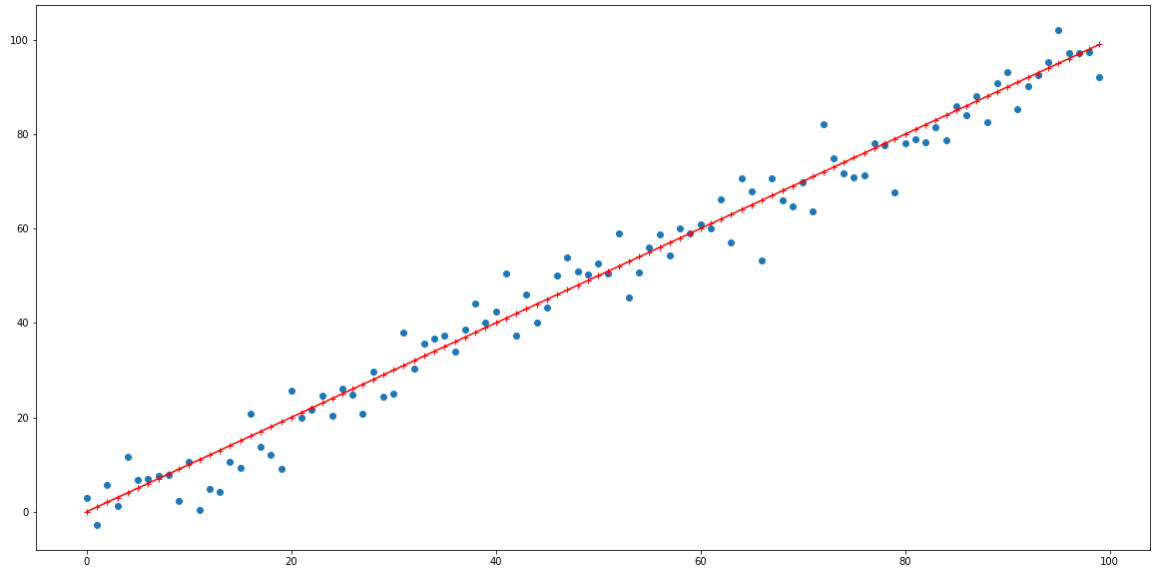
Out[ ]:

```
array([  3.03517893,  -2.82117412,   5.63641171,   1.22718468,
        11.53161315,   6.85036967,   7.02046492,   7.51948556,
         7.90080738,   2.30278501,  10.46915097,   0.46101934,
         4.75528817,   4.13542247,  10.60353609,   9.31862581,
        20.68257291,  13.79460557,  12.10555955,   9.12471064,
        25.55078676,  19.93659549,  21.54255645,  24.59587454,
        20.357809  ,  26.15390175,  24.72052946,  20.84819328,
        29.67710055,  24.38538656,  24.93568762,  38.01629247,
        30.26770206,  35.54897579,  36.60704533,  37.32712445,
        33.9165728 ,  38.6949391 ,  44.13655623,  40.04854594,
        42.38126495,  50.44758534,  37.270093  ,  46.09763445,
        40.08030826,  43.23792946,  49.97387177,  53.84766767,
        50.97640145,  50.30539833,  52.59859053,  50.57769671,
        58.88362419,  45.43095789,  50.70294935,  56.08892189,
        58.84146136,  54.39778624,  59.9306173 ,  59.01749713,
        60.88173735,  60.04802837,  66.15927616,  56.99173842,
        70.72263381,  67.94630284,  53.16207429,  70.74283331,
        65.89754152,  64.76443263,  69.85256696,  63.58859571,
        82.212065  ,  74.87262769,  71.75124002,  70.79231424,
        71.32882562,  78.01681038,  77.59572503,  67.75286821,
        77.97428603,  78.89755941,  78.32339119,  81.5427152 ,
        78.63630454,  85.96496243,  83.9391429 ,  87.96975109,
        82.4667936 ,  90.73401437,  93.19535762,  85.34434596,
        90.26998726,  92.494145  ,  95.22880342, 102.08225711,
        97.17475512,  97.19108117,  97.4393878 ,  92.17161386])
```

We can show both the new points with the old function together in the same graph.

In [ ]:

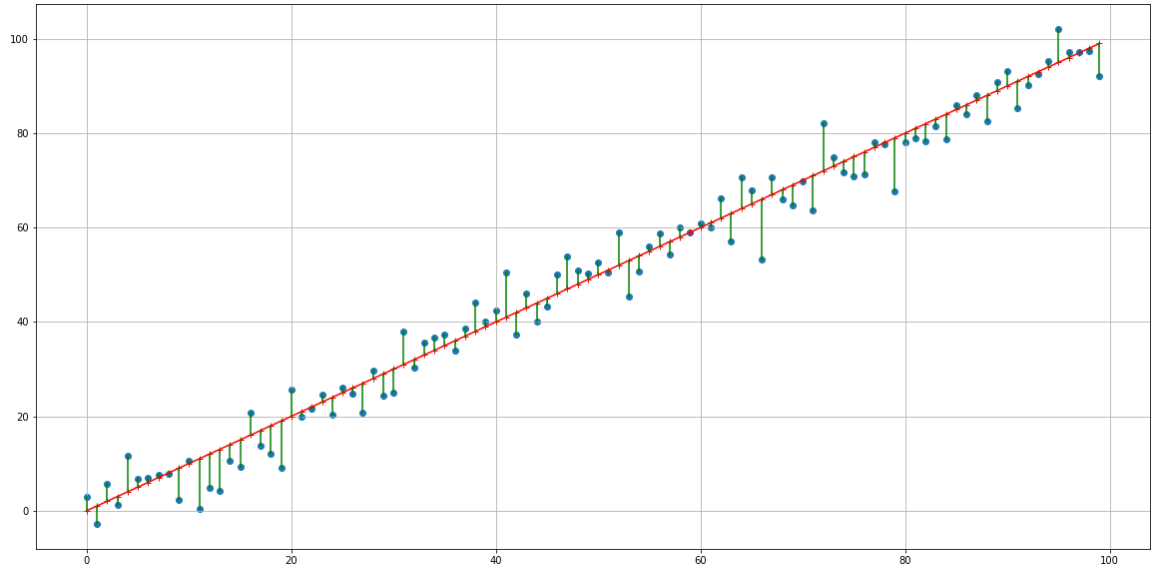
```
# show graph of our original function and the new points
plt.scatter(x, new_y)
plt.plot(x, y, '+-r')
plt.show()
```



We want to find out how much the new points are far from the original function.
To do so, we need to develop a way of evaluating this value.
First, let's show the distance of each point from its original position.

In [ ]:

```
# add lines that represent the distance of each noisy point from the original point
plt.scatter(x, new_y)
plt.plot(x, y, '+-r')
for px, py, pny in zip(x, y, new_y):
    plt.plot([px, px], [py, pny], "-g")
plt.grid()
plt.show()
```



We can sum all the distances and get a number that represents the error of all the points, but then, with more points, we will have a bigger error value (we sum more values).
We want to get a value that will give a similar value range with fewer or more points.
So, we can divide the summation by the number of points and get the mean error.
The only thing we have to decide on is the way we calculate the distance.
This distance should be positive, no matter where is the noisy point in the graph (above or below the function line).
So, we have 2 options:

1. Take the distance and raise it to the power of 2 to get `dist^2`. This approach is called **MSE - Mean Squared Error**.
2. Take the absolute value of the distance to get `|dist|`. This approach is called **MAE - Mean Absolute Error**.

You can read more on the differences between them in [MAE and RMSE — Which Metric is Better? (https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d)](https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d)
(RMSE is the squared root of the MSE, it is very similar and for our usage, there is no difference in using one over the other).

Let's use the MSE approach.

In [ ]:

```
# calculate MSE of the points in the graph
sum_dist = 0
for px, py, pny in zip(x, y, new_y):
    sum_dist += (py - pny)**2
mean_dist = sum_dist/len(y)
mean_dist
```

Out[ ]:

```
20.967311769524564
```

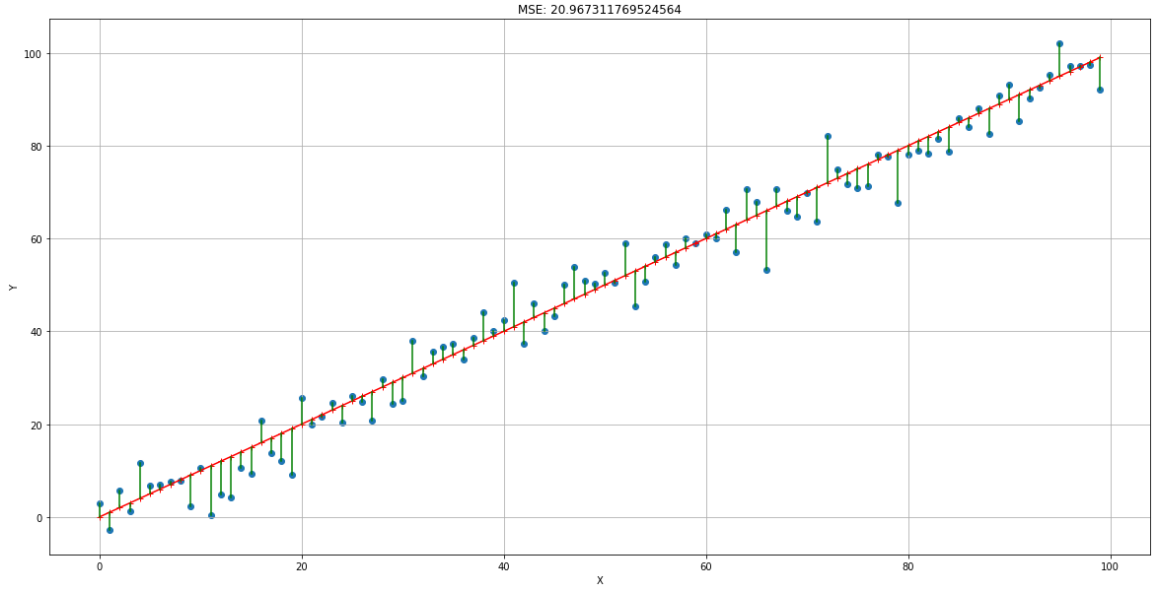
We can define methods that will help us automate this process and create graphs for any function.

In [ ]:

```
# show graph of x, y and new_y values
def show_graph(x, y, new_y):
    plt.scatter(x, new_y)
    plt.plot(x, y, '+-r')
    for px, py, pny in zip(x, y, new_y):
        plt.plot([px, px], [py, pny], "-g")
    plt.grid()
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title(f'MSE: {calculate_mse(y, new_y)}')

# calculate the mse for x, y and new_y values
def calculate_mse(y, new_y):
    sum_dist = 0
    for py, pny in zip(y, new_y):
        sum_dist += (py - pny)**2
    mse = sum_dist/len(y)
    return mse

show_graph(x, y, new_y)
```

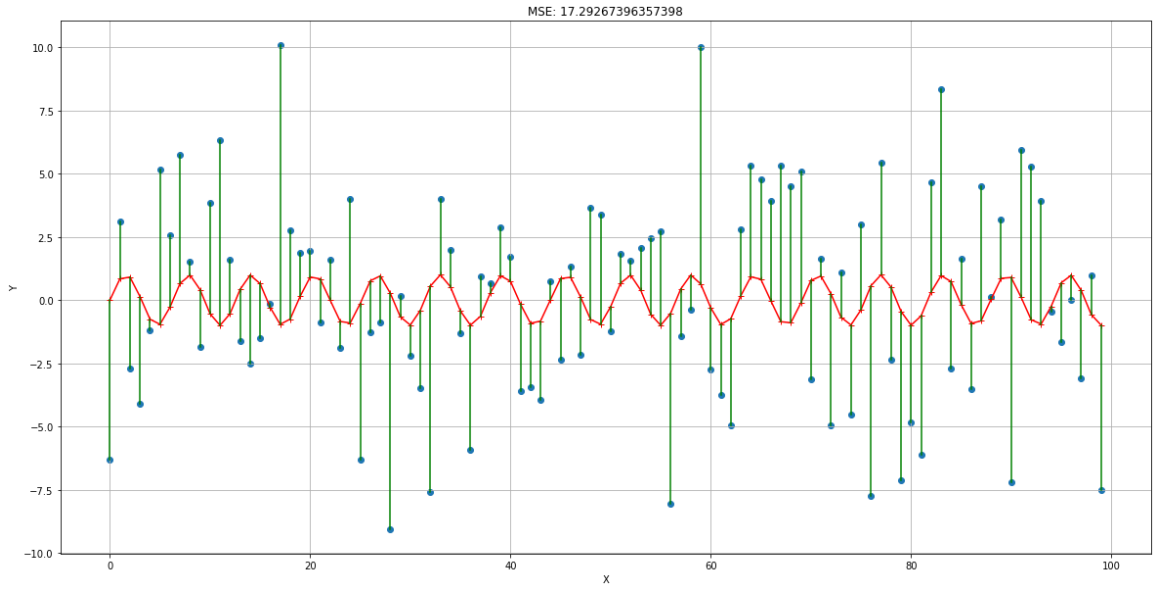


We can also define a method that will create the new noisy points from the original points.
We can use NumPy `sin` method to create the function `y(x) = sin(x)` .

In [ ]:

```
# create noisy y from original y
def create_new_y_from_y(y, sd):
    return y + np.random.normal(loc=0, scale=sd, size=[len(y)])

y = np.sin(x)
new_y = create_new_y_from_y(y, 5)
show_graph(x, y, new_y)
```



ScyPy

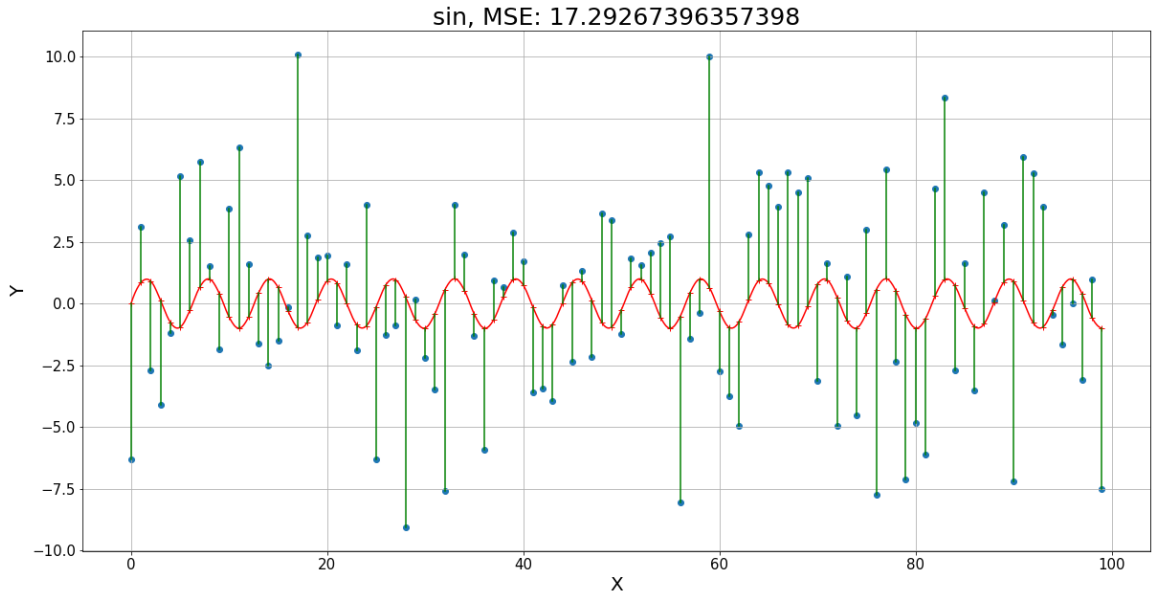
We have a problem with this graph, the lines of the `sin` function are not rounded enough. It is just a bunch of red lines and it is not resembling much of the `sin` function we know.
We can use ScyPy interpolation to make the lines and their connections smoother.

In [ ]:

```
# import `interp1d` module form `scipy.iterpolate` library and show the graph with the
 interpolation
from scipy.interpolate import interp1d

def show_graph_with_interpolation(x, y, new_y, title):
    xnew = np.linspace(x.min(), x.max(), num=len(x)*10, endpoint=True)
    f3 = interp1d(x, y, kind='quadratic')
    plt.plot(xnew, f3(xnew), '-r')
    plt.scatter(x, new_y)
    plt.plot(x, y, '+r')
    for px, py, pny in zip(x, y, new_y):
        plt.plot([px, px], [py, pny], "-g")
    plt.grid()
    plt.xlabel('X', fontsize=20)
    plt.ylabel('Y', fontsize=20)
    plt.title(f'{title}, MSE: {calculate_mse(y, new_y)}', fontsize=25)
    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)

show_graph_with_interpolation(x, y, new_y, 'sin')
```



Let's define a method that will create any function we want (well.., we need to create the support for them in this method, but let's pretend we can create any function we want).

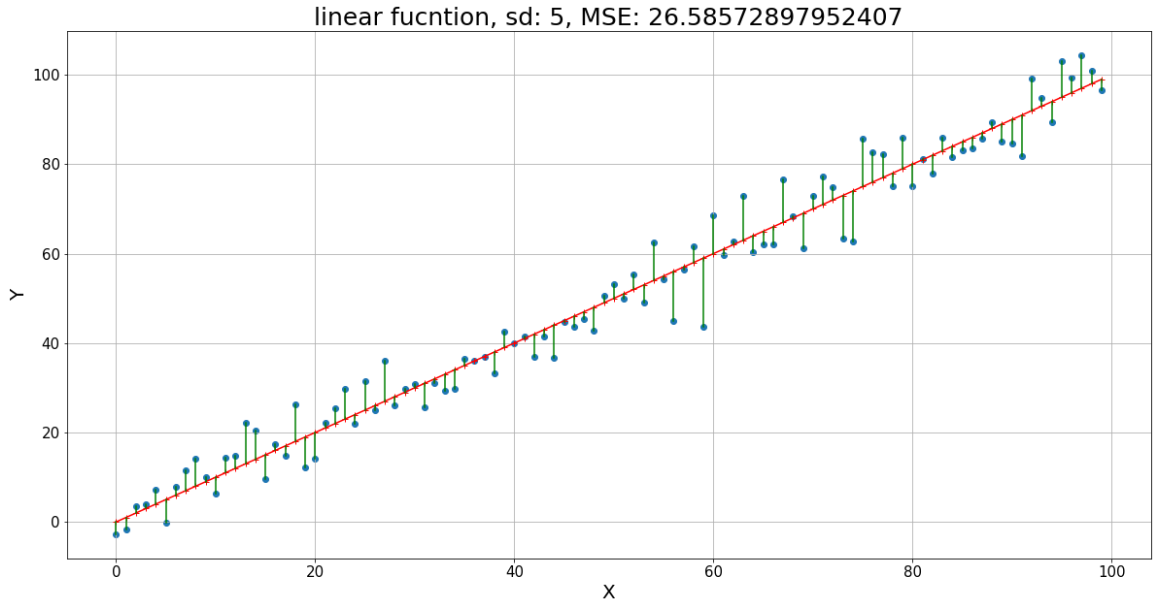
In [ ]:

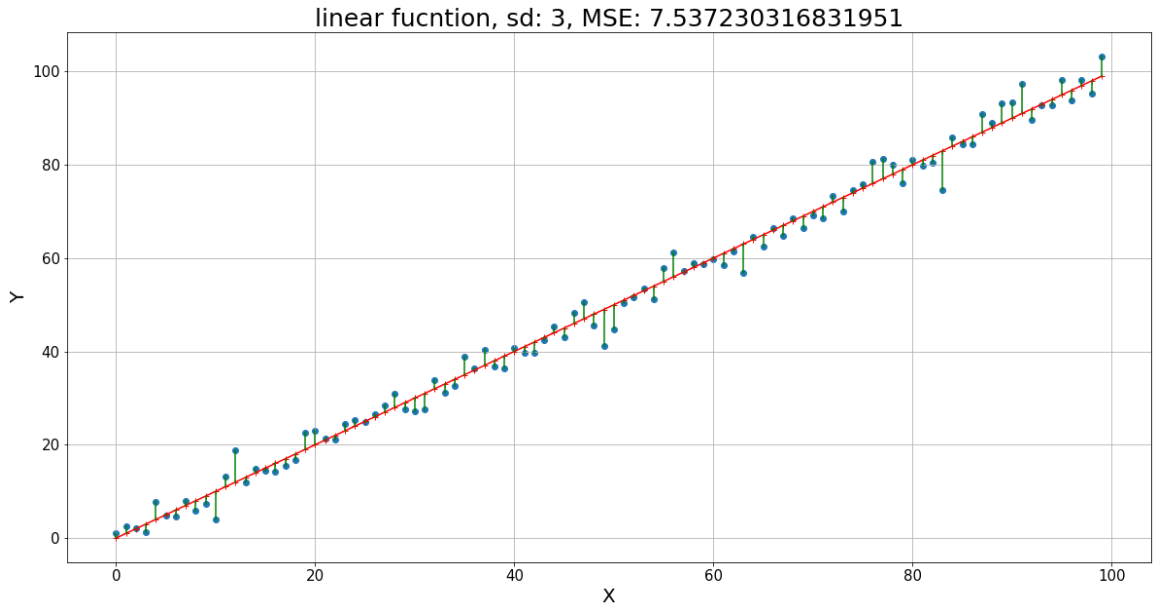
```
# showes the function with the number of points specified
# we can change the standard deviation of the noisy points
def plot_func_and_rand_points(x, func, sd, func_name='????'):
    y = func(x)
    new_y = create_new_y_from_y(y, sd)
    show_graph_with_interpolation(x, y, new_y, f'{func_name} fucntion, sd: {sd}')
```

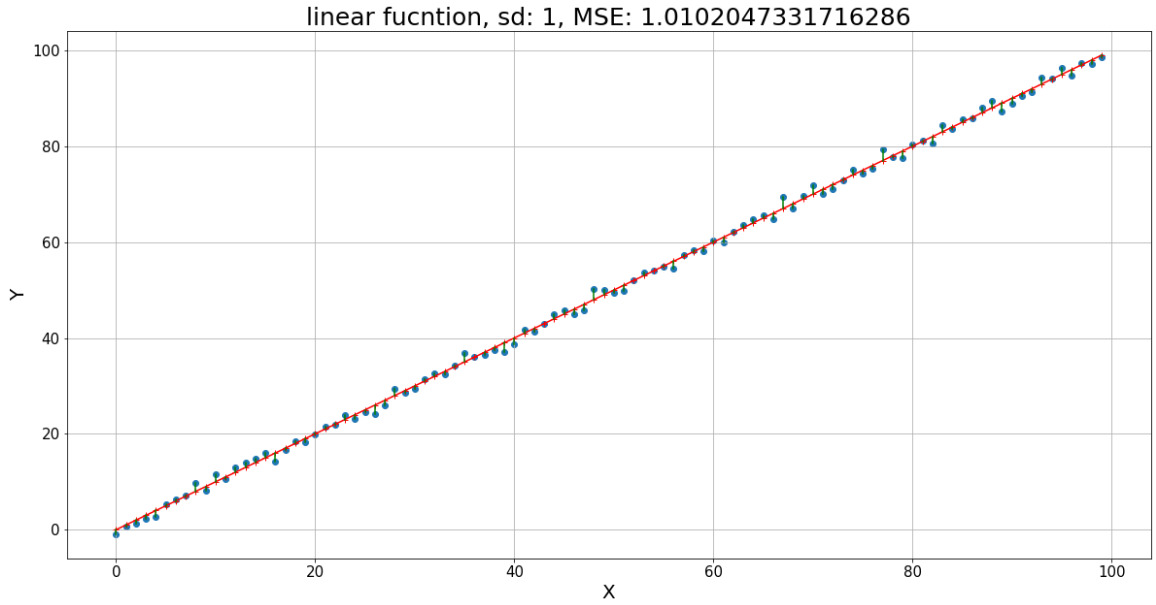
Now, we can investigate the effect of the standard deviation on the MSE.
Let's try to insert different `sd` for different functions and see how does the MSE changes.

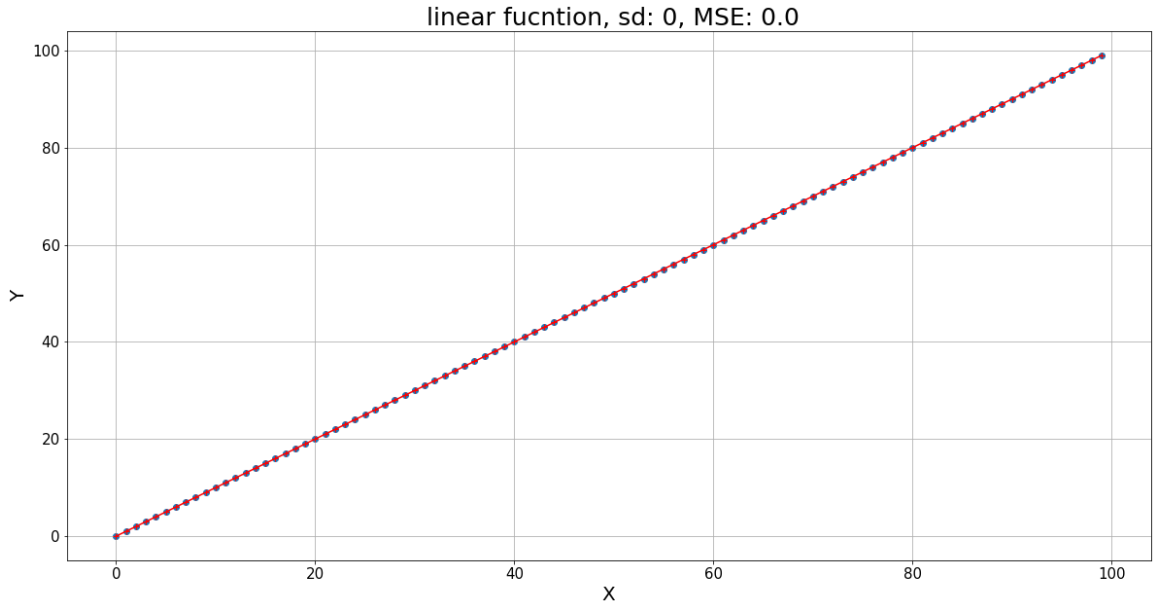
In [ ]:

```
# show sd=[5, 3, 1, 0] for linear function
plot_func_and_rand_points(np.arange(100), lambda x: x.copy(), 5, 'linear')
plt.show()
plot_func_and_rand_points(np.arange(100), lambda x: x.copy(), 3, 'linear')
plt.show()
plot_func_and_rand_points(np.arange(100), lambda x: x.copy(), 1, 'linear')
plt.show()
plot_func_and_rand_points(np.arange(100), lambda x: x.copy(), 0,' linear')
plt.show()
```


linear fucntion, sd: 5, MSE: 26.58572897952407
Y
X
0
20
40
60
80
100

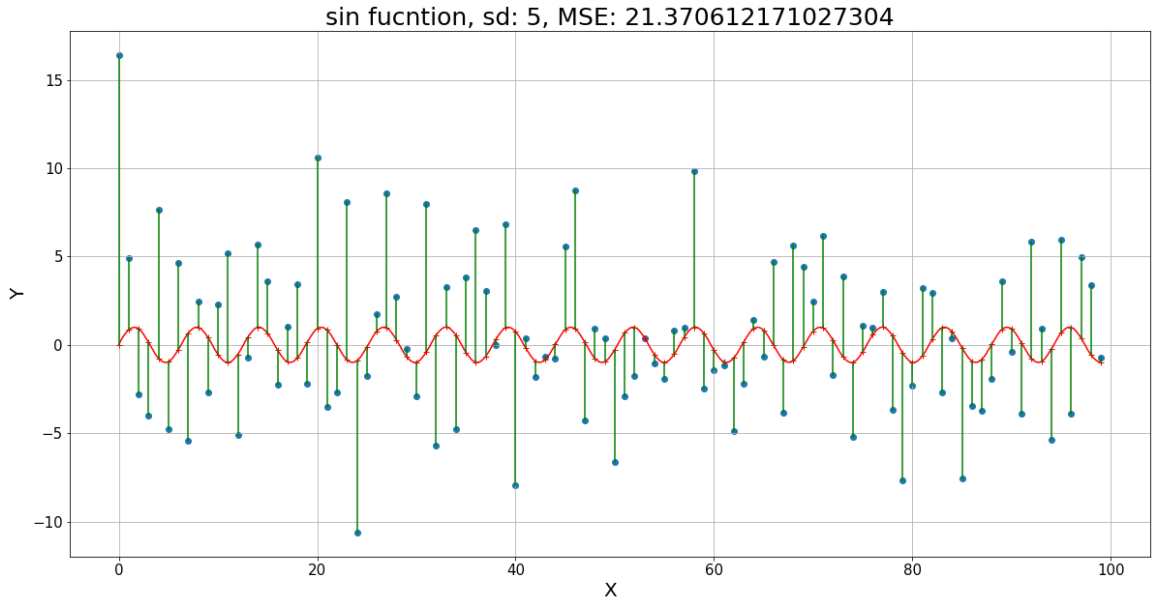

linear fucntion, sd: 3, MSE: 7.537230316831951
Y
X
0
20
40
60
80
100

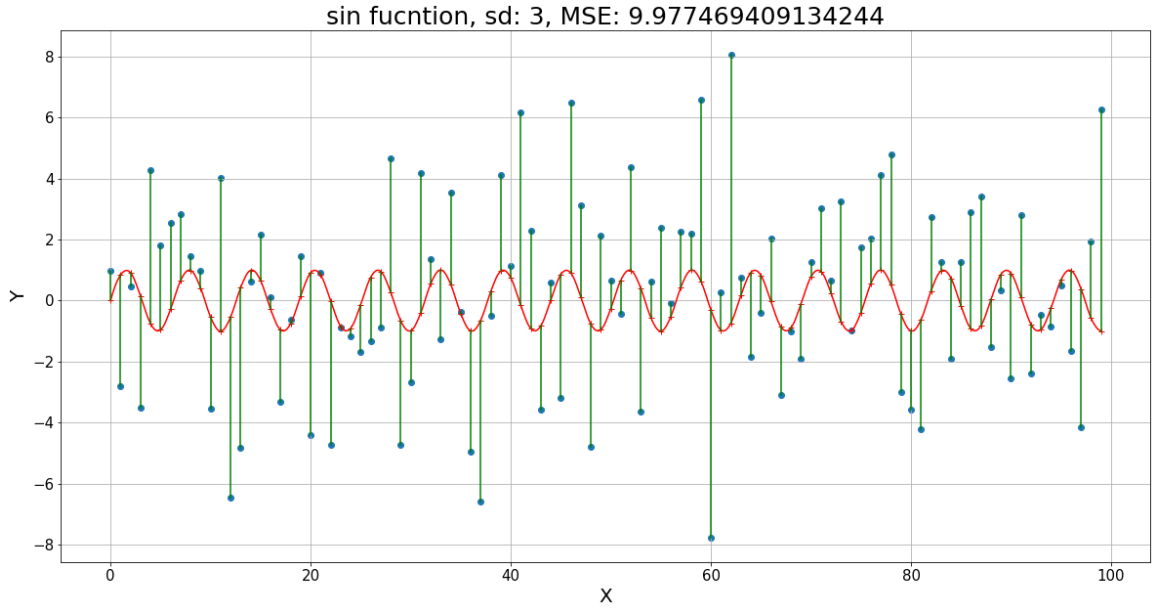

linear fucntion, sd: 1, MSE: 1.0102047331716286
Y
X
0
20
40
60
80
100

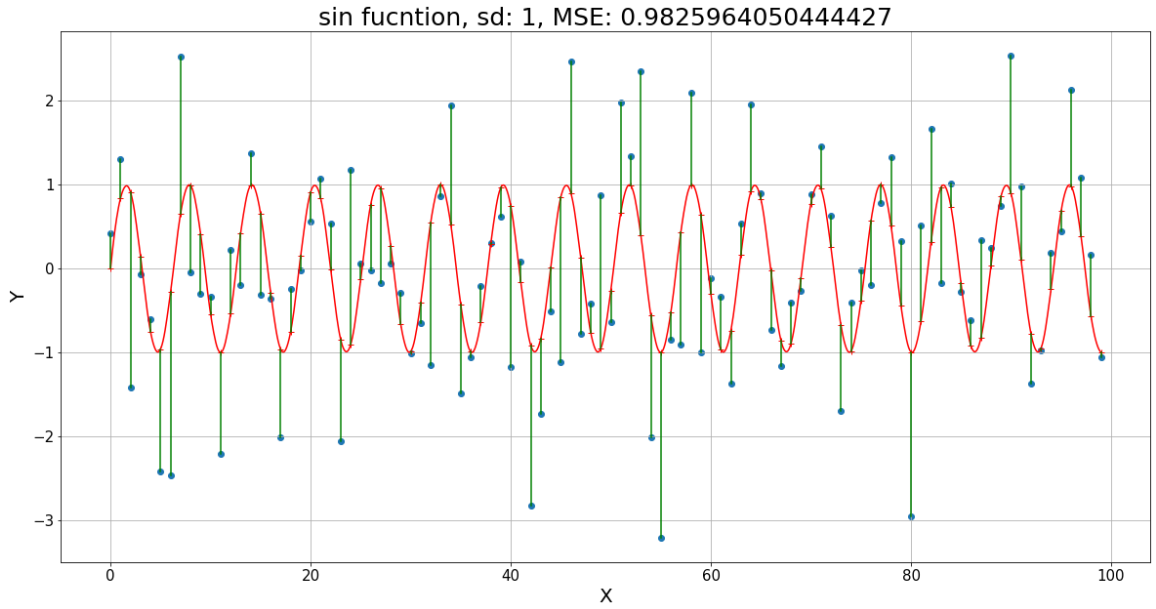

linear fucntion, sd: 0, MSE: 0.0
Y
X
0
20
40
60
80
100

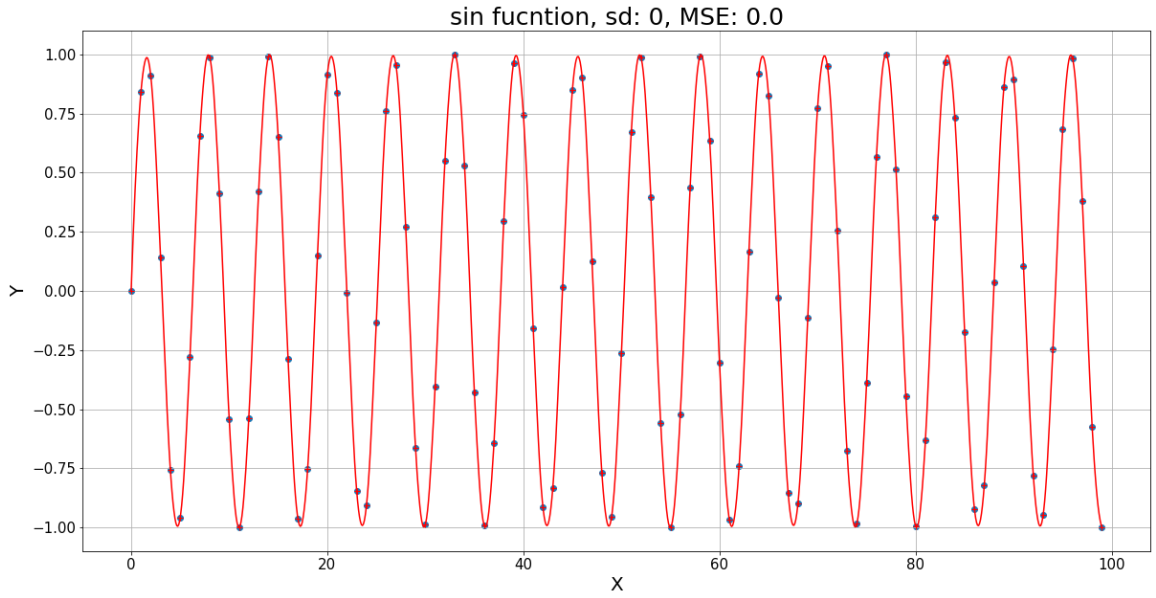
In [ ]:

```
# show sd=[5, 3, 1, 0] for sin function
plot_func_and_rand_points(np.arange(100), np.sin, 5, 'sin')
plt.show()
plot_func_and_rand_points(np.arange(100), np.sin, 3, 'sin')
plt.show()
plot_func_and_rand_points(np.arange(100), np.sin, 1, 'sin')
plt.show()
plot_func_and_rand_points(np.arange(100), np.sin, 0, 'sin')
plt.show()
```


sin fucntion, sd: 5, MSE: 21.370612171027304
Y
X

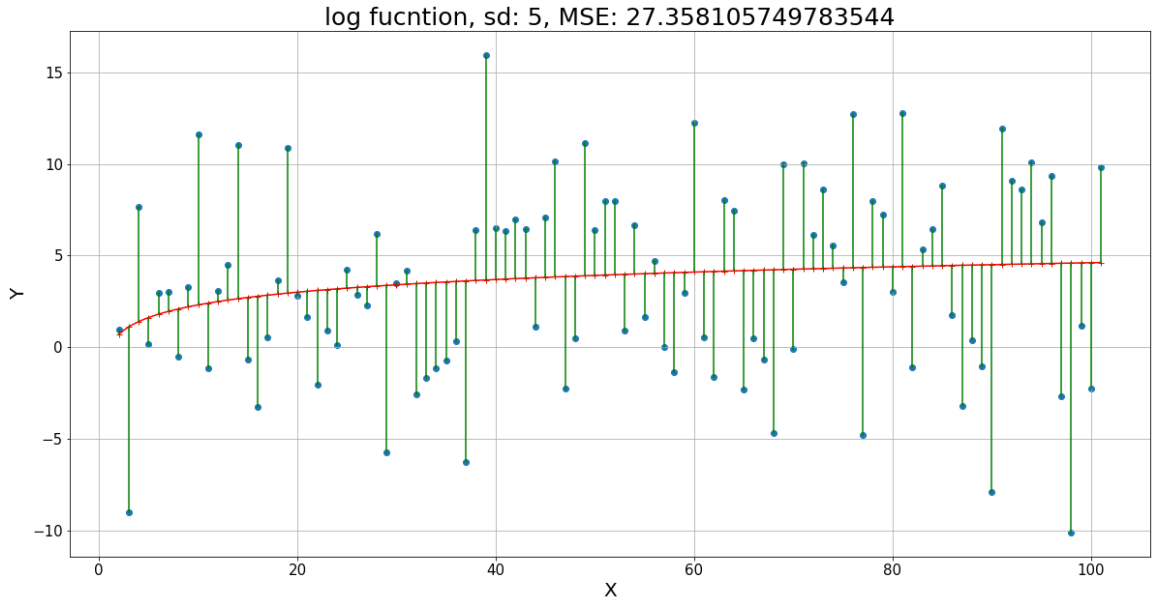

sin fucntion, sd: 3, MSE: 9.977469409134244
Y
X

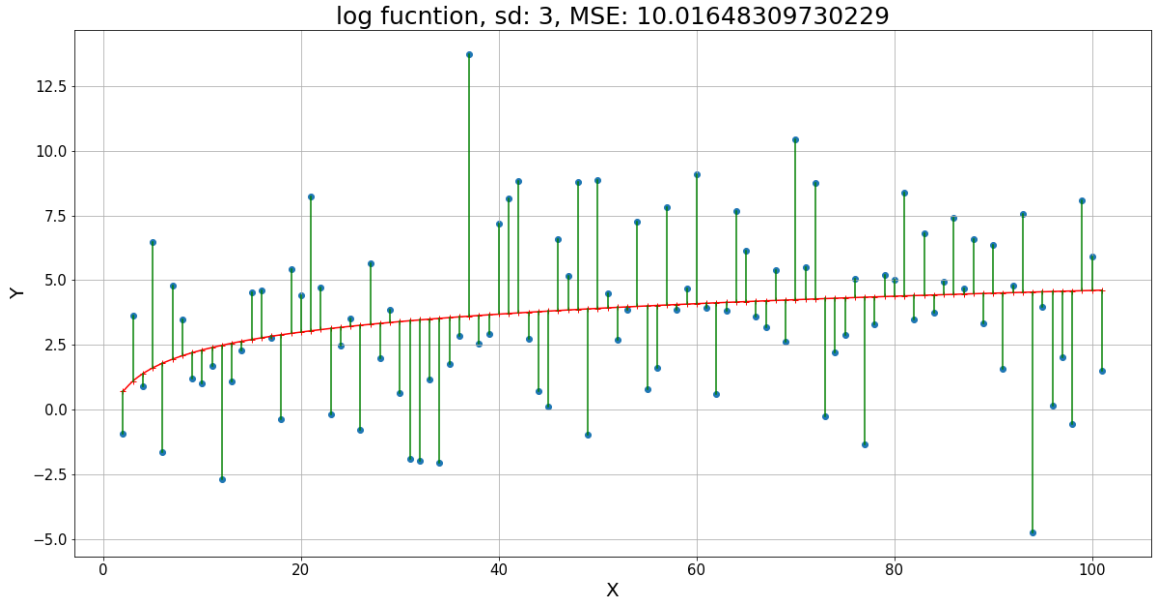

sin fucntion, sd: 1, MSE: 0.9825964050444427
Y
X

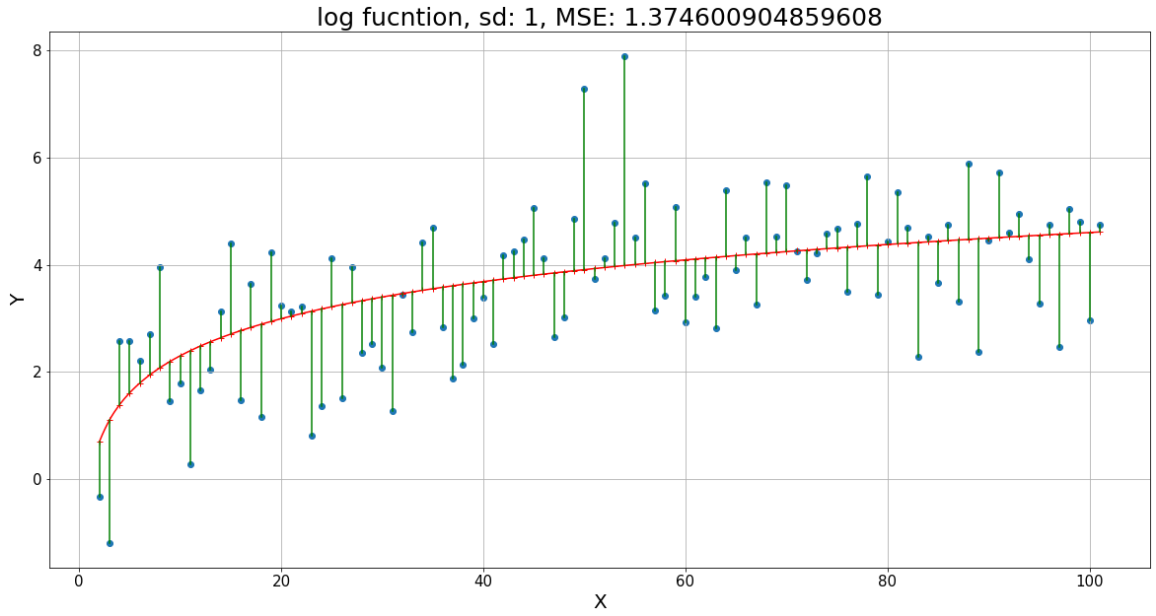

sin fucntion, sd: 0, MSE: 0.0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Y
0
20
40
60
80
100
X

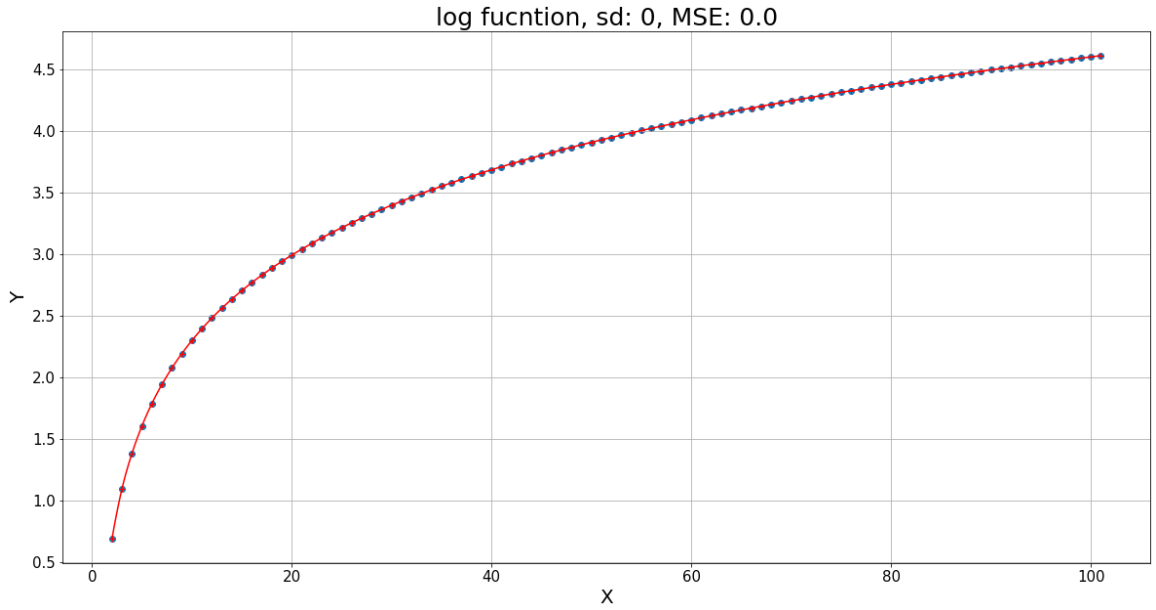
In [ ]:

```
# show sd=[5, 3, 1, 0] for log function
plot_func_and_rand_points(np.arange(2, 102), np.log, 5, 'log')
plt.show()
plot_func_and_rand_points(np.arange(2, 102), np.log, 3, 'log')
plt.show()
plot_func_and_rand_points(np.arange(2, 102), np.log, 1, 'log')
plt.show()
plot_func_and_rand_points(np.arange(2, 102), np.log, 0, 'log')
plt.show()
```


log fucntion, sd: 5, MSE: 27.358105749783544
Y
X


log fucntion, sd: 3, MSE: 10.01648309730229
Y
X





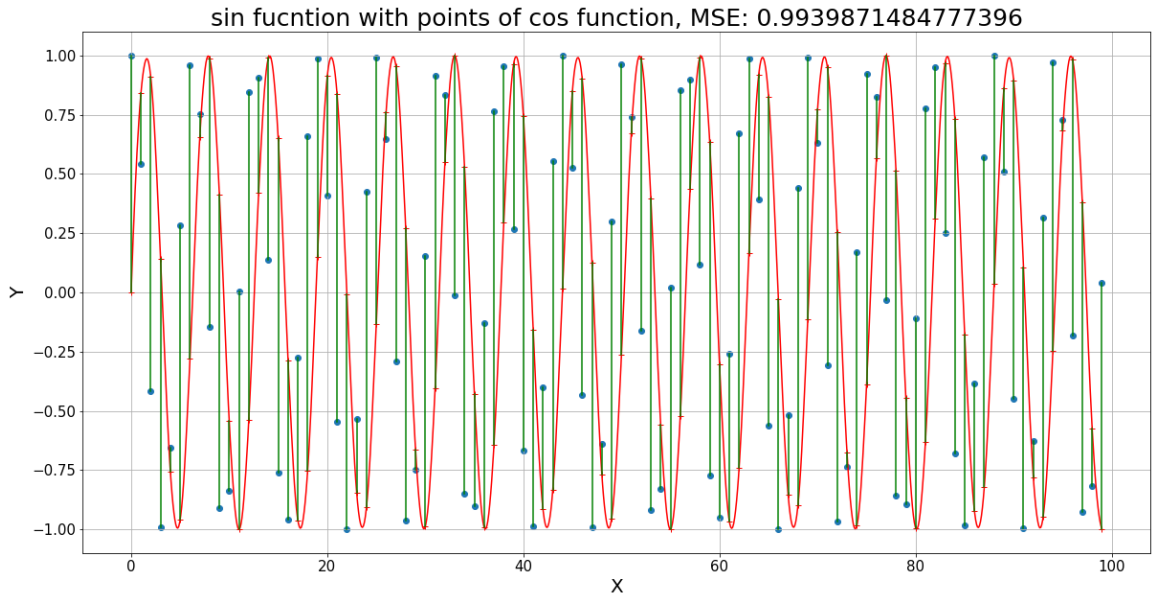
We can see that the bigger the `sd`, the bigger the MSE.
When the `sd` is `0`, the MSE is `0` too (all the noisy points are the same as the original points - there is no noise\error).
We can also check, what will happen if we will try to calculate the MSE between original points of one function and noisy points of other function.

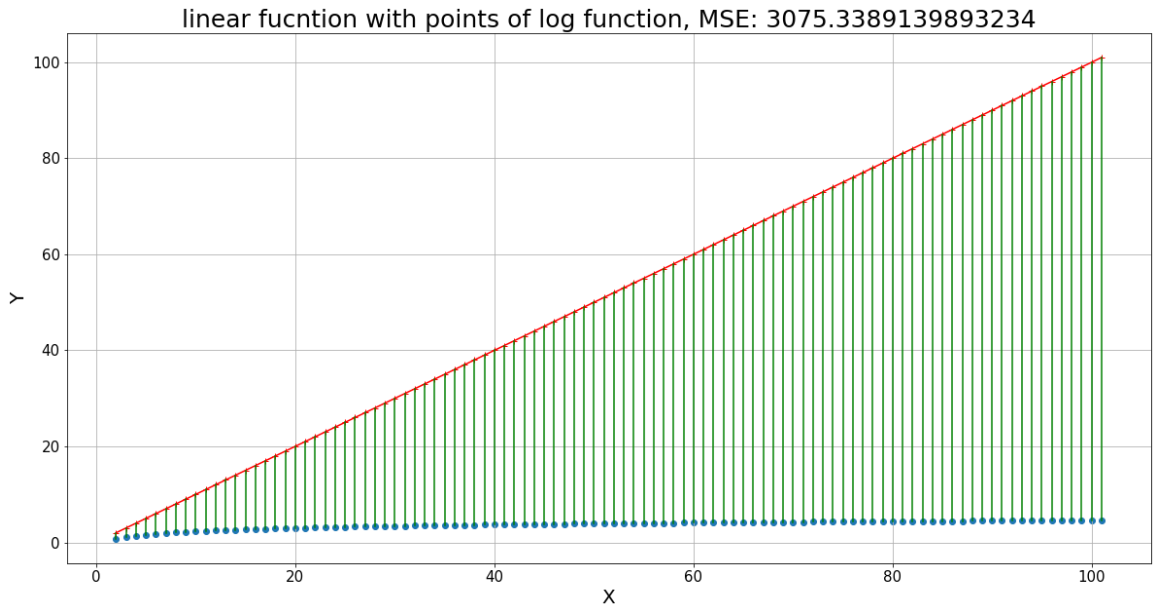
In [ ]:

```
# showes both functions with number of points specified and calculate MSE
# for log, we take points from 2 and not from 0 because log(0) is infinity and
# log(1) is 0
def plot_func_and_rand_points_two_functions(x, func1, func2, func1_name='????', func2_n
ame='????'):
    y1 = func1(x)
    y2 = func2(x)
    show_graph_with_interpolation(x, y1, y2, f'{func1_name} fucntion with points of {fu
nc2_name} function')
```

In [ ]:

```
# show MSE of some combinations of functions
plot_func_and_rand_points_two_functions(np.arange(100), np.sin, np.cos, 'sin', 'cos')
plt.show()
plot_func_and_rand_points_two_functions(np.arange(2, 102), lambda x: x.copy(), np.log,
'linear', 'log')
plt.show()
```



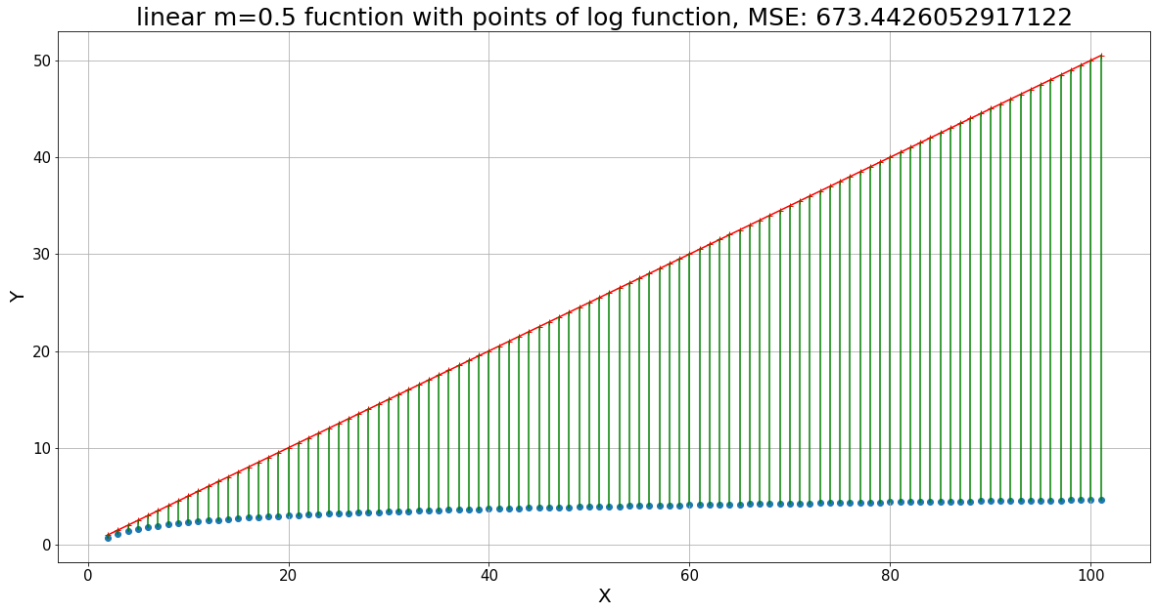


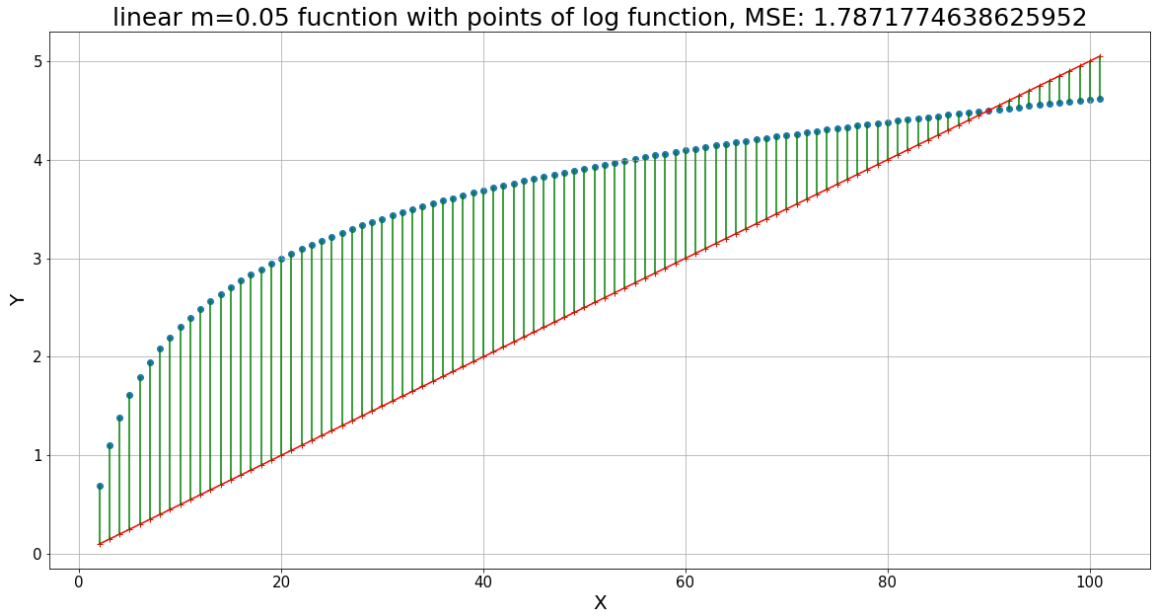
We can see that sometimes the MSE is big and sometimes it is not so bad.
We can also see that even if in the sin-cos combination, the functions are different, the MSE is low because the values are low. There are no points that lie very far from the function line.
In the linear-log combination, the points are very far from the function line. That makes the MSE huge.

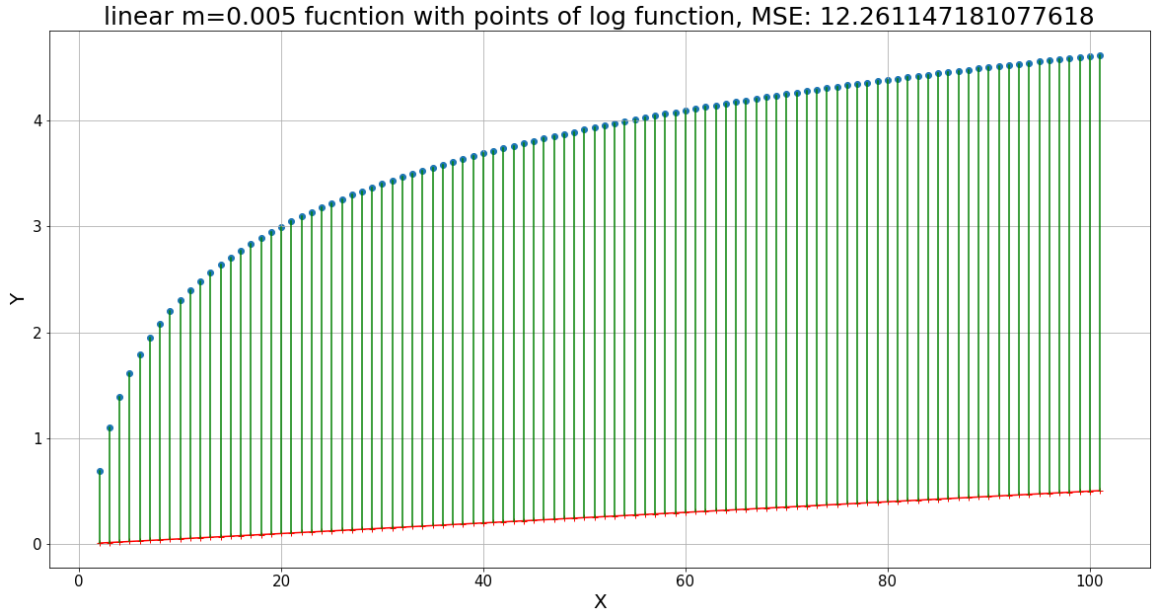
We can try to reduce the linear-log MSE, by changing the slope of the linear function.

In [ ]:

```
# show MSE of some combinations of `linear` (m=[0.5, 0.05, 0.005]) and `log`
plot_func_and_rand_points_two_functions(np.arange(2, 102), lambda x: 0.5*x.copy(), np.l
og, 'linear m=0.5', 'log')
plt.show()
plot_func_and_rand_points_two_functions(np.arange(2, 102), lambda x: 0.05*x.copy(), np.
log, 'linear m=0.05', 'log')
plt.show()
plot_func_and_rand_points_two_functions(np.arange(2, 102), lambda x: 0.005*x.copy(), np
.log, 'linear m=0.005', 'log')
plt.show()
```


linear m=0.5 fucntion with points of log function, MSE: 673.4426052917122
Y
X
0
10
20
30
40
50
0
20
40
60
80
100


linear m=0.05 fucntion with points of log function, MSE: 1.7871774638625952
Y
X
0
1
2
3
4
5
0
20
40
60
80
100


linear m=0.005 fucntion with points of log function, MSE: 12.261147181077618
Y
X
0
1
2
3
4
0
20
40
60
80
100

We can write the slopes and the MSE to try to figure out what is the best slope that gives us the lowest MSE:

slope	1	0.5	0.05	0.005
MSE	3075	673	1	12

We can see that at first, the MSE is getting lower along with the slope, but this trend flips when we get to `slope=0.05` and `slope=0.005`.
The MSE is suddenly getting larger.
It means that the best MSE value for these functions is lying somewhere between `slope=0.5` and `slope=0.005`. The best slope in our attempts is `slope=1` but we can not know if it is the best one or maybe there are better slopes out there (a hint, something out there may look better, but is it worth the trouble?).

SymPy

Let's see the definition of the MSE expression we created in mathematics.
To do this we use the SymPy library.
We need to build the expression like a lego from building blocks.
We first define the smallest building blocks (symbols, indexes, functions, etc.).
Then, we can build the expression from these blocks.
The expressions are rendered with Unicode (https://en.wikipedia.org/wiki/Unicode).

In order to be compatible with the literature on the subject and with the lecture, let's change the variables.
From now on, the variable we called `original_y` or `y`, will be `t` as of *true values*. These are the true values of the points.
The variables that we called `noisy_y` or `new_y` will be just `y`. This is the function that we try to fit the true values.
The variable `x` will stay `x`.

In [ ]:

```
# import all the moduls inside `sympy` library and create the basic building blocks for our expression
from sympy import *
n, i = symbols('n i')
xi = Indexed('x', i)
ti = Indexed('t', i)
yf = Function('y')
init_printing(use_unicode=True)
```

In [ ]:

```
# create the MSE expression from the basic building blocks
MSE = (1/n)*Sum((ti-yf(xi))**2, (i, 1, n))
MSE
```

Out[ ]:

$$\frac{1}{n}\sum_{i=1}^{n}\left(-y(x_i)+t_i\right)^2$$

We can also differentiate the MSE expression with SymPy.

In [ ]:

```
# get the derivative of MSE
MSE_dif = diff(MSE, xi)
MSE_dif
```

Out[ ]:

$$\frac{1}{n}\sum_{i=1}^{n} -2\left(-y(x_i)+t_i\right)\frac{\partial}{\partial x_i}y(x_i)$$

Let's see the string representation of the above expression.

In [ ]:

```
# show the string representation of MSE expression in sympy
srepr(MSE)
```

Out[ ]:

```
"Mul(Pow(Symbol('n'), Integer(-1)), Sum(Pow(Add(Mul(Integer(-1), Function
('y')(Indexed(IndexedBase(Symbol('x')), Symbol('i')))), Indexed(IndexedBas
e(Symbol('t')), Symbol('i'))), Integer(2)), Tuple(Symbol('i'), Integer(1),
Symbol('n'))))"
```

We can also show the graph of this expression.
It is a graph with nodes and edges, that is in a tree form (https://en.wikipedia.org/wiki/Tree_(graph_theory)).

In [ ]:

```
# show the graph as a string with pprint (pretty printing)
import pprint
from sympy.printing.dot import dotprint

pprint.pprint(dotprint(MSE))
```

```
('digraph{\n'
 '\n'
 '# Graph style\n'
 '"ordering"="out"\n'
 '"rankdir"="TD"\n'
 '\n'
 '#########\n'
 '# Nodes #\n'
 '#########\n'
 '\n'
 '"Mul(Pow(Symbol(n), NegativeOne()), Sum(Pow(Add(Mul(NegativeOne(), '
 'y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2)), Tuple(Symbol
(i), '
 'One(), Symbol(n))))_()" ["color"="black", "label"="Mul", '
 '"shape"="ellipse"];\n'
 '"Pow(Symbol(n), NegativeOne())_(0,)" ["color"="black", "label"="Pow", '
 '"shape"="ellipse"];\n'
 '"Symbol(n)_(0, 0)" ["color"="black", "label"="n", "shape"="ellipse"];\n'
 '"NegativeOne()_(0, 1)" ["color"="black", "label"="-1", "shape"="ellips
e"];\n'
 '"Sum(Pow(Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), '
 'Symbol(i)))), Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2)),
'
 'Tuple(Symbol(i), One(), Symbol(n)))_(1,)" ["color"="black", "label"="Su
m", '
 '"shape"="ellipse"];\n'
 '"Pow(Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol
(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2))_(1, 0)" '
 '["color"="black", "label"="Pow", "shape"="ellipse"];\n'
 '"Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))),
'
 'Indexed(IndexedBase(Symbol(t)), Symbol(i)))_(1, 0, 0)" ["color"="black",
'
 '"label"="Add", "shape"="ellipse"];\n'
 '"Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i))))_(1,
0, 0, '
 '0)" ["color"="black", "label"="Mul", "shape"="ellipse"];\n'
 '"NegativeOne()_(1, 0, 0, 0, 0)" ["color"="black", "label"="-1", '
 '"shape"="ellipse"];\n'
 '"y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))_(1, 0, 0, 0, 1)" '
 '["color"="black", "label"="y", "shape"="ellipse"];\n'
 '"Indexed(IndexedBase(Symbol(x)), Symbol(i))_(1, 0, 0, 0, 1, 0)" '
 '["color"="black", "label"="x[i]", "shape"="ellipse"];\n'
 '"IndexedBase(Symbol(x))_(1, 0, 0, 0, 1, 0, 0)" ["color"="black", '
 '"label"="x", "shape"="ellipse"];\n'
 '"Symbol(x)_(1, 0, 0, 0, 1, 0, 0, 0)" ["color"="black", "label"="x", '
 '"shape"="ellipse"];\n'
 '"Symbol(i)_(1, 0, 0, 0, 1, 0, 1)" ["color"="black", "label"="i", '
 '"shape"="ellipse"];\n'
 '"Indexed(IndexedBase(Symbol(t)), Symbol(i))_(1, 0, 0, 1)" ["color"="blac
k", '
 '"label"="t[i]", "shape"="ellipse"];\n'
 '"IndexedBase(Symbol(t))_(1, 0, 0, 1, 0)" ["color"="black", "label"="t",
'
 '"shape"="ellipse"];\n'
 '"Symbol(t)_(1, 0, 0, 1, 0, 0)" ["color"="black", "label"="t", '
 '"shape"="ellipse"];\n'
 '"Symbol(i)_(1, 0, 0, 1, 1)" ["color"="black", "label"="i", '
 '"shape"="ellipse"];\n'
```

```
 '"Integer(2)_(1, 0, 1)" ["color"="black", "label"="2", "shape"="ellips
e"];\n'
 '"Tuple(Symbol(i), One(), Symbol(n))_(1, 1)" ["color"="blue", '
 '"label"="Tuple", "shape"="ellipse"];\n'
 '"Symbol(i)_(1, 1, 0)" ["color"="black", "label"="i", "shape"="ellips
e"];\n'
 '"One()_(1, 1, 1)" ["color"="black", "label"="1", "shape"="ellipse"];\n'
 '"Symbol(n)_(1, 1, 2)" ["color"="black", "label"="n", "shape"="ellips
e"];\n'
 '\n'
 '#########\n'
 '# Edges #\n'
 '#########\n'
 '\n'
 '"Mul(Pow(Symbol(n), NegativeOne()), Sum(Pow(Add(Mul(NegativeOne(), '
 'y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2)), Tuple(Symbol
(i), '
 'One(), Symbol(n))))_()" -> "Pow(Symbol(n), NegativeOne())_(0,)";\n'
 '"Mul(Pow(Symbol(n), NegativeOne()), Sum(Pow(Add(Mul(NegativeOne(), '
 'y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2)), Tuple(Symbol
(i), '
 'One(), Symbol(n))))_()" -> "Sum(Pow(Add(Mul(NegativeOne(), '
 'y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2)), Tuple(Symbol
(i), '
 'One(), Symbol(n)))_(1,)";\n'
 '"Pow(Symbol(n), NegativeOne())_(0,)" -> "Symbol(n)_(0, 0)";\n'
 '"Pow(Symbol(n), NegativeOne())_(0,)" -> "NegativeOne()_(0, 1)";\n'
 '"Sum(Pow(Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), '
 'Symbol(i)))), Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2)),
'
 'Tuple(Symbol(i), One(), Symbol(n)))_(1,)" -> "Pow(Add(Mul(NegativeOne(),
'
 'y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2))_(1, 0)";\n'
 '"Sum(Pow(Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), '
 'Symbol(i)))), Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2)),
'
 'Tuple(Symbol(i), One(), Symbol(n)))_(1,)" -> "Tuple(Symbol(i), One(), '
 'Symbol(n))_(1, 1)";\n'
 '"Pow(Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol
(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2))_(1, 0)" -> '
 '"Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))),
'
 'Indexed(IndexedBase(Symbol(t)), Symbol(i)))_(1, 0, 0)";\n'
 '"Pow(Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol
(i)))), '
 'Indexed(IndexedBase(Symbol(t)), Symbol(i))), Integer(2))_(1, 0)" -> '
 '"Integer(2)_(1, 0, 1)";\n'
 '"Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))),
'
 'Indexed(IndexedBase(Symbol(t)), Symbol(i)))_(1, 0, 0)" -> '
 '"Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i))))_(1,
0, 0, '
 '0)";\n'
 '"Add(Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))),
'
 'Indexed(IndexedBase(Symbol(t)), Symbol(i)))_(1, 0, 0)" -> '
```

```
 '"Indexed(IndexedBase(Symbol(t)), Symbol(i))_(1, 0, 0, 1)";\n'
 '"Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i))))_(1,
0, 0, '
 '0)" -> "NegativeOne()_(1, 0, 0, 0, 0)";\n'
 '"Mul(NegativeOne(), y(Indexed(IndexedBase(Symbol(x)), Symbol(i))))_(1,
0, 0, '
 '0)" -> "y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))_(1, 0, 0, 0,
1)";\n'
 '"y(Indexed(IndexedBase(Symbol(x)), Symbol(i)))_(1, 0, 0, 0, 1)" -> '
 '"Indexed(IndexedBase(Symbol(x)), Symbol(i))_(1, 0, 0, 0, 1, 0)";\n'
 '"Indexed(IndexedBase(Symbol(x)), Symbol(i))_(1, 0, 0, 0, 1, 0)" -> '
 '"IndexedBase(Symbol(x))_(1, 0, 0, 0, 1, 0, 0)";\n'
 '"Indexed(IndexedBase(Symbol(x)), Symbol(i))_(1, 0, 0, 0, 1, 0)" -> '
 '"Symbol(i)_(1, 0, 0, 0, 1, 0, 1)";\n'
 '"IndexedBase(Symbol(x))_(1, 0, 0, 0, 1, 0, 0)" -> "Symbol(x)_(1, 0, 0,
0, 1, '
 '0, 0, 0)";\n'
 '"Indexed(IndexedBase(Symbol(t)), Symbol(i))_(1, 0, 0, 1)" -> '
 '"IndexedBase(Symbol(t))_(1, 0, 0, 1, 0)";\n'
 '"Indexed(IndexedBase(Symbol(t)), Symbol(i))_(1, 0, 0, 1)" -> "Symbol(i)_
(1, '
 '0, 0, 1, 1)";\n'
 '"IndexedBase(Symbol(t))_(1, 0, 0, 1, 0)" -> "Symbol(t)_(1, 0, 0, 1, 0,
0)";\n'
 '"Tuple(Symbol(i), One(), Symbol(n))_(1, 1)" -> "Symbol(i)_(1, 1, 0)";\n'
 '"Tuple(Symbol(i), One(), Symbol(n))_(1, 1)" -> "One()_(1, 1, 1)";\n'
 '"Tuple(Symbol(i), One(), Symbol(n))_(1, 1)" -> "Symbol(n)_(1, 1, 2)";\n'
 '}')
```

We want to see the graph in a more interpretable visualization.
We can do it with the `graphviz` library.
We render the graph in **SVG** format (https://he.wikipedia.org/wiki/SVG).
This will save the graph image sharp in any size.

In [ ]:

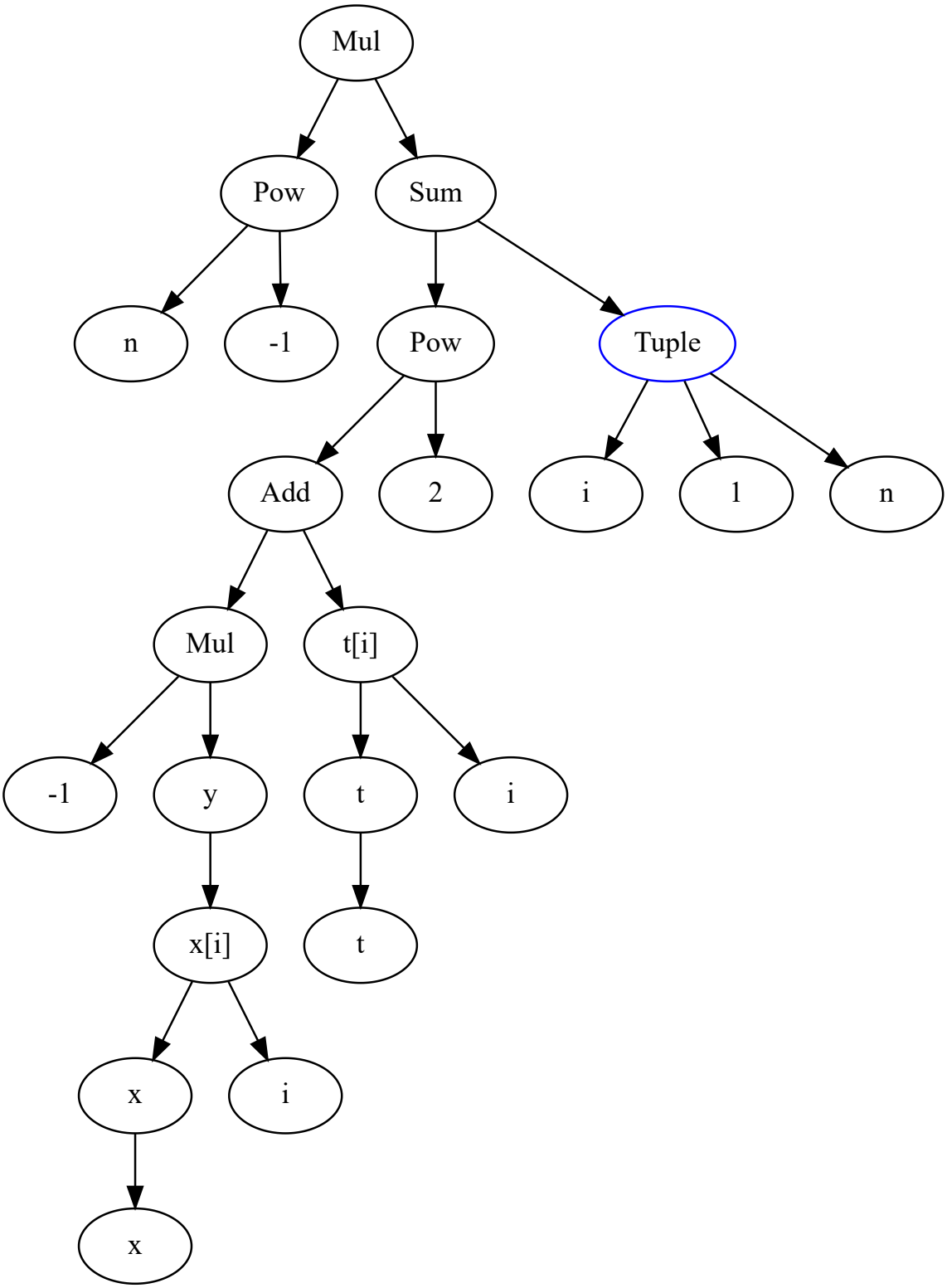
```
# import `Source` module form `graphviz` library and use it to graph the tree
# save it in SVG and display it
from graphviz import Source
from IPython.display import SVG
import os

def draw_tree(expr, delete_file=True, file_name='temp'):
    src = Source(dotprint(expr))
    src.render(file_name, format='svg')
    svg = SVG(filename=(f'{file_name}.svg'))
    os.unlink(f'{file_name}.svg')
    return svg

display(MSE)
draw_tree(MSE)
```

$$\frac{1}{n}\sum_{i=1}^{n}\left(-y(x_i)+t_i\right)^2$$

Out[ ]:



Let's calculate the MSE value from our points and functions.
To do so, we will define a method that represents a function that gets `x` and `t` (original points) and returns the noisy points.
We will transfer this `y` function with the `x` and `t` variables to the `mse` method, to calculate the MSE from the SymPy expression.

In [ ]:

```
# get the `y_func` we want to evaluate in the MSE expression
def get_y_func(func_name, sd):
    def inner_func(x):
        SUPPORTED_FUNCTIONS = ['linear', 'sin', 'cos', 'exp']

        if func_name == 'linear':
            y = lambda x: x
        elif func_name == 'sin':
            y = np.sin
        elif func_name == 'cos':
            y = np.cos
        elif func_name == 'exp':
            y = np.exp
        else:
            raise ValueError(f'func_name variable must be one of {SUPPORTED_FUNCTIONS}.
\n\
                you enterd `{func_name}`')

        eps = np.random.normal(scale=sd)
        return y(x) + eps

    return inner_func

# evaluate the MSE of the `y_func` on the data points (x, t)
def mse(x, t, y_func):
    n = symbols('n')
    xi = Indexed('x', i)
    ti = Indexed('t', i)
    yf = Function('y')
    MSE = (1/n)*Sum((ti-yf(xi))**2, (i, 1, n))
    one_on_n = MSE.args[0].subs({n: len(x)})
    sum_i_to_n = MSE.args[1].subs({n: len(x)})
    y = list(map(y_func, x))
    dist_pow_2 = (t - y)**2
    return MSE.func(one_on_n.doit(), Subs(sum_i_to_n.doit(), [sum_i_to_n.function.subs(
sum_i_to_n.variables[0], j) for j in range(sum_i_to_n.limits[0][1], sum_i_to_n.limits[0
][2] + 1)], dist_pow_2).doit()).doit()
```

Let's use the methods we created to calculate the MSE of 100 points that were created from the `sin` function (with noise).

In [ ]:

```
# create the original points that will be used in MSE
x = np.arange(100)
t = np.sin(x)

# create the function to enter the mse
y_func = get_y_func('sin', 5)

# calculate the mse
mse(x, t, y_func)
```

Out[ ]:

22.6385955561118

More Information

MSE on Wikipedia:
Wikipedia MSE (https://en.wikipedia.org/wiki/Mean_squared_error)

An explanation of NumPy axes:
NumPy Axes Explained (https://www.sharpsightlabs.com/blog/numpy-axes-explained/)

Documentation of NumPy library:
NumPy Documentation (https://numpy.org/doc/stable/reference/routines.html)

Documentation of matplotlib.pyplot.scatter method:
Matplotlib pyplot.scatter Documentation
(https://matplotlib.org/3.3.1/api/_as_gen/matplotlib.pyplot.scatter.html)

Documentation of matplotlib.pyplot.plot method:
Matplotlib pyplot.plot Documentation (https://matplotlib.org/3.3.1/api/_as_gen/matplotlib.pyplot.plot.html)

Documentation of plt.rcParams dictionary parameters:
Customizing Matplotlib with style sheets and rcParams
(https://matplotlib.org/3.3.2/tutorials/introductory/customizing.html)

Tutorial for matplotlib.pyplot (`plt`) module:
Matplotlib pyplot Tutorial (https://matplotlib.org/tutorials/introductory/pyplot.html).

Tutorial for SymPy library:
SymPy Tutorial (http://certik.github.io/scipy-2013-tutorial/html/tutorial/index.html)

Why there are so many ways to import libraries:
Official abbreviation for: import scipy as sp/sc (https://stackoverflow.com/questions/36014733/official-abbreviation-for-import-scipy-as-sp-sc).

There is another way of displaying the tree graph in IPython:
render_sympy_tree.py (https://gist.github.com/goerz/178811cece8e96ae5378be71b8ea904a).

SymPy subscript:
How do I define a sympy symbol with a subscript string?
(https://stackoverflow.com/questions/24897931/how-do-i-define-a-sympy-symbol-with-a-subscript-string)

Markers in Matplotlib plot line:
Set markers for individual points on a line in Matplotlib (https://stackoverflow.com/questions/8409095/set-markers-for-individual-points-on-a-line-in-matplotlib)

Site for resizing images:
resizeimage.net (https://resizeimage.net/)