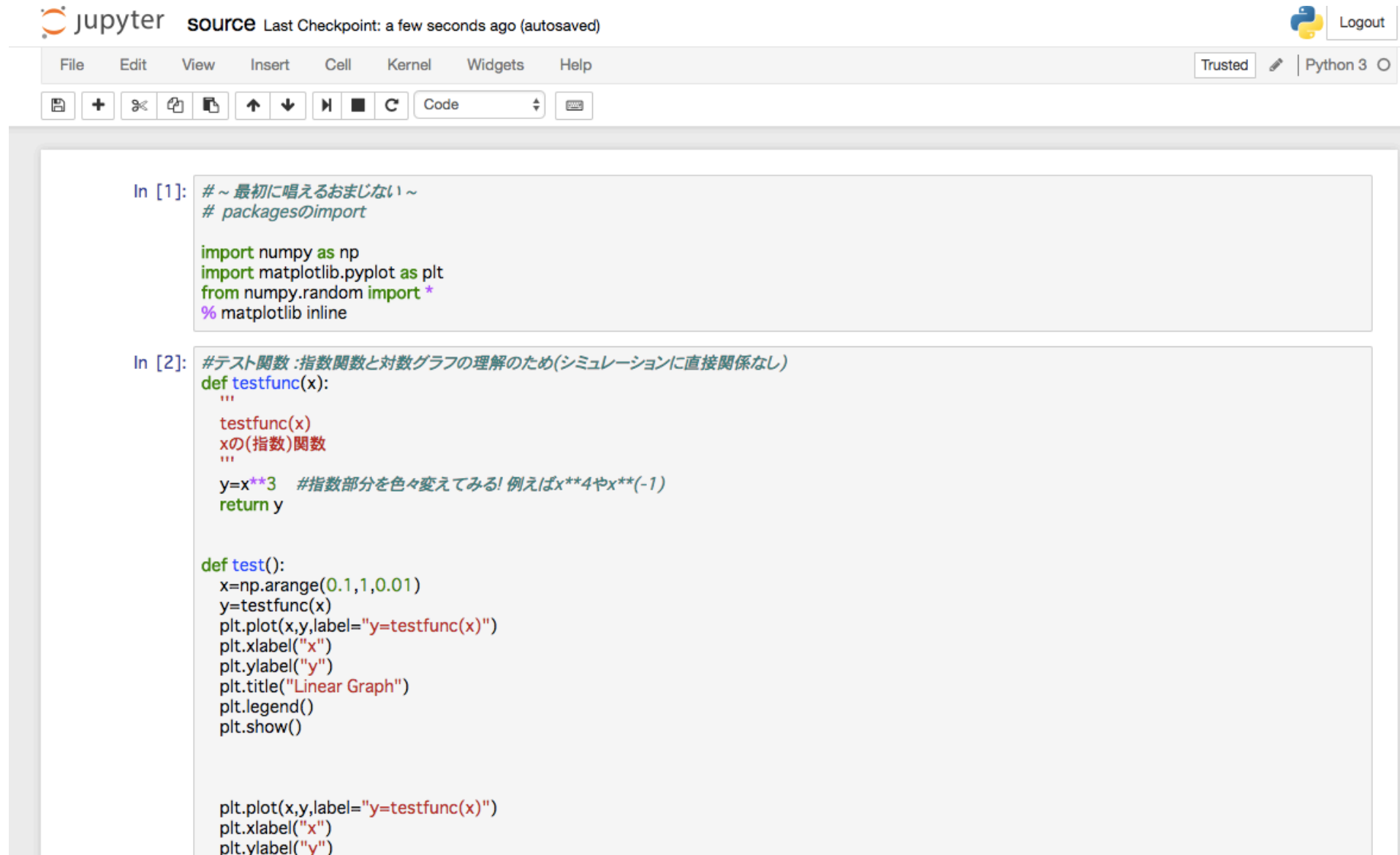


実習

プログラムファイルを開く

Jupyter Notebookを起動した後に、配布したフォルダ内のsource.ipynbというファイルを開く

このような画面が出ればOK



The screenshot shows the Jupyter Notebook web interface. At the top, the header includes the Jupyter logo, the word "source", and a status message "Last Checkpoint: a few seconds ago (autosaved)". On the right, there is a "Logout" button and a Python 3 logo. Below the header is a menu bar with "File", "Edit", "View", "Insert", "Cell", "Kernel", "Widgets", and "Help". A toolbar below the menu bar contains icons for saving, adding new cells, undo, redo, and other actions. The main area displays two code cells. The first cell, labeled "In [1]:", contains import statements for numpy, matplotlib, and random. The second cell, labeled "In [2]:", contains a function definition for testfunc, a test function, and plotting code using matplotlib.

```
In [1]: # ~ 最初に唱えるおまじない ~  
# packagesのimport  
  
import numpy as np  
import matplotlib.pyplot as plt  
from numpy.random import *  
% matplotlib inline  
  
In [2]: #テスト関数 :指数関数と対数グラフの理解のため(シミュレーションに直接関係なし)  
def testfunc(x):  
    """  
    testfunc(x)  
    xの(指数)関数  
    """  
    y=x**3 #指数部分を色々変えてみる! 例えばx**4やx**(-1)  
    return y  
  
def test():  
    x=np.arange(0.1,1,0.01)  
    y=testfunc(x)  
    plt.plot(x,y,label="y=testfunc(x)")  
    plt.xlabel("x")  
    plt.ylabel("y")  
    plt.title("Linear Graph")  
    plt.legend()  
    plt.show()  
  
plt.plot(x,y,label="y=testfunc(x)")  
plt.xlabel("x")  
plt.ylabel("y")
```

プログラムの大枠

ヘッダー + 関数の定義 + Main関数 でできている

ヘッダー：おまじない(わかっていなくて良い)

関数の定義: Main関数の中で使う関数を定義(defから始まる)

ヘッダー

関数の定義

```
In [1]: # ~ 最初に唱えるおまじない ~
# packagesのimport

import numpy as np
import matplotlib.pyplot as plt
from numpy.random import *
% matplotlib inline

In [2]: # テスト関数: 指数関数と対数グラフの理解のためのシミュレーションに直接関係なし
def testfunc(x):
    """
    testfunc(x)
    xの(指数)関数
    """
    y=x**3 # 指数部分を色々変えてみる! 例えばx**4やx**(-1)
    return y

def test():
    x=np.arange(0.1,1,0.01)
    y=testfunc(x)
    plt.plot(x,y,label="y=testfunc(x)")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Linear Graph")
    plt.legend()
    plt.show()

plt.plot(x,y,label="y=testfunc(x)")
plt.xlabel("x")
plt.ylabel("y")
```

プログラムの大枠

ヘッダー + 関数の定義 + Main関数 でできている

Main関数: Mainのプログラム. ここで指示した順に実行される.

Main関数 →

```
jupyter source Last Checkpoint: 31 minutes ago (autosaved) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
[54]: %%time
#####
#シミュレーションの設定(次数mの完全グラフを初期状態にとる)
#####

# パラメータ: ここを変えてみて結果がどう変わるか見てみよう!
m=4 # 追加するnodeの持つlinkの本数
Nmax=100 # 追加するnodeの個数
a=0. # 重み付け確率を計算する時のパラメータ

# 初期状態を生成する
# node数mの完全グラフを作成
nw=comgra(m)

#####
#ネットワークを成長させる
#####
# Nmax回m本の枝を持つnodeを追加する
for j in range(0,Nmax):
    # 関数calcでネットワークnwにnodeを追加していく
    # 3個目の引数(重み付け確率でシミュレーション:type=1, 一様確率でシミュレーション:type=2)
    calc(nw,a,m,type=1) # typeの値を変えてみよう!

np.save("nwdata.npy",nw)

#####
#分布を計算する
#####
dist(nw,m)
# x.npyには次数の列が入っている
# y.npyには数値シミュレーションした各次数の頻度が入っている
# y2.npyには理論値の各次数の頻度が入っている

CPU times: user 16.8 ms, sys: 2.11 ms, total: 18.9 ms
Wall time: 18.3 ms
```

プログラムの大枠

ヘッダー + 関数の定義 + Main関数 でできている

Main関数の中で、定義した関数を使ってやりたいことを順番に命令する

ヘッダー

関数の定義

関数の定義

関数の定義

Main関数

Mainプログラムの内容

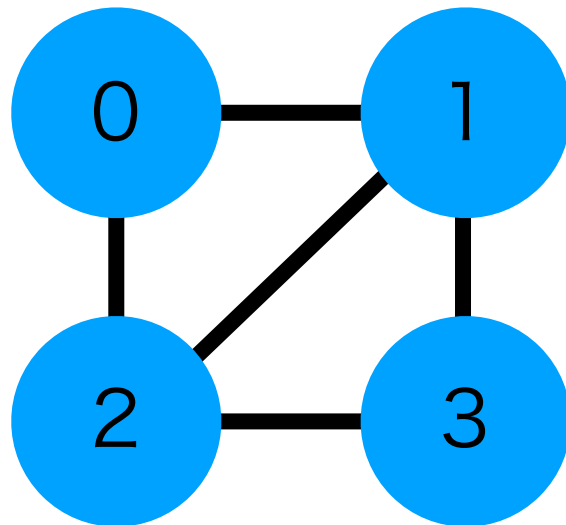
- ▷ Main関数の中身がわかれば大体何をしているかわかる
- ▷ 大事な変数は「nw」: ネットワークを表現するデータ配列

隣接リスト:

[(node0に繋がっているnodeからなるlist), (node1に繋がっているnodeからなるlist),...]

例

[[1,2], [0,2,3], [0,1,3], [1,2]]



これ!



```
jupyter source Last Checkpoint: 31 minutes ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
[Icons] Code
In [54]: %%time
#####
#シミュレーションの設定(次数mの完全グラフを初期状態にとる)
#####

# パラメータ: ここを変えてみて結果がどう変わるか見てみよう!
m=4 # 追加するnodeの持つlinkの本数
Nmax=100 # 追加するnodeの個数
a=0. # 重み付け確率を計算する時のパラメータ

# 初期状態を生成する
# node数mの完全グラフを作成
nw=comgra(m)

#####
#ネットワークを成長させる
#####
# Nmax回m本の枝を持つnodeを追加する
for j in range(0,Nmax):
    # 関数calcでネットワークnwにnodeを追加していく
    # 3個目の引数(重み付け確率でシミュレーション:type=1, 一様確率でシミュレーション:type=2)
    calc(nw,a,m,type=1) # typeの値を変えてみよう!

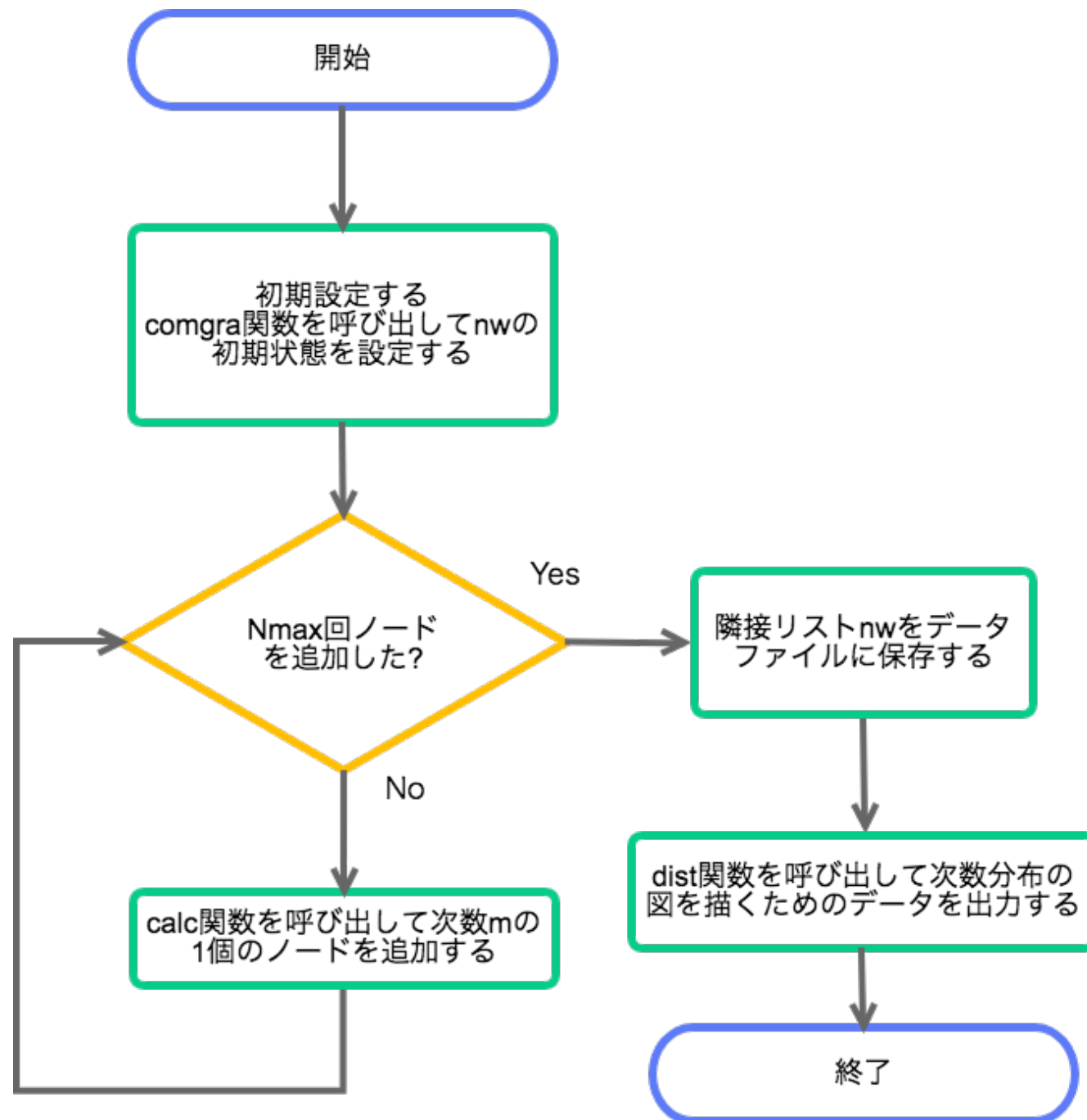
np.save("nwdata.npy",nw)

#####
#分布を計算する
```


Mainプログラムのアルゴリズム

▷ Main関数の中身がわかれば大体何をしているかわかる

フローチャートとプログラムを比較して理解する



jupyter source Last Checkpoint: 31 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

In [54]: `%%time`

```
#####  
#シミュレーションの設定(次数mの完全グラフを初期状態にとる)  
#####  
  
# パラメータ：ここを変えてみて結果がどう変わるか見てみよう!  
m=4 # 追加するnodeの持つlinkの本数  
Nmax=100 # 追加するnodeの個数  
a=0. # 重み付け確率を計算する時のパラメータ  
  
# 初期状態を生成する  
# node数mの完全グラフを作成  
nw=comgra(m)  
  
#####  
#ネットワークを成長させる  
#####  
# Nmax回m本の枝を持つnodeを追加する  
for j in range(0,Nmax):  
    # 関数calcでネットワークnwにnodeを追加していく  
    # 3個目の引数(重み付け確率でシミュレーション:type=1,一様確率  
    calc(nw,a,m,type=1) # typeの値を変えてみよう!  
  
np.save("nwdata.npy",nw)  
  
#####  
#分布を計算する  
#####  
dist(nw,m)  
# x.npyには次数の列が入っている  
# y.npyには数値シミュレーションした各次数の頻度が入っている  
# y2.npyには理論値の各次数の頻度が入っている
```

CPU times: user 16.8 ms, sys: 2.11 ms, total: 18.9 ms
Wall time: 18.3 ms

定義した関数の内容

def 関数名(引数)

...

return 返回值

引数: $y=f(x)$ の x

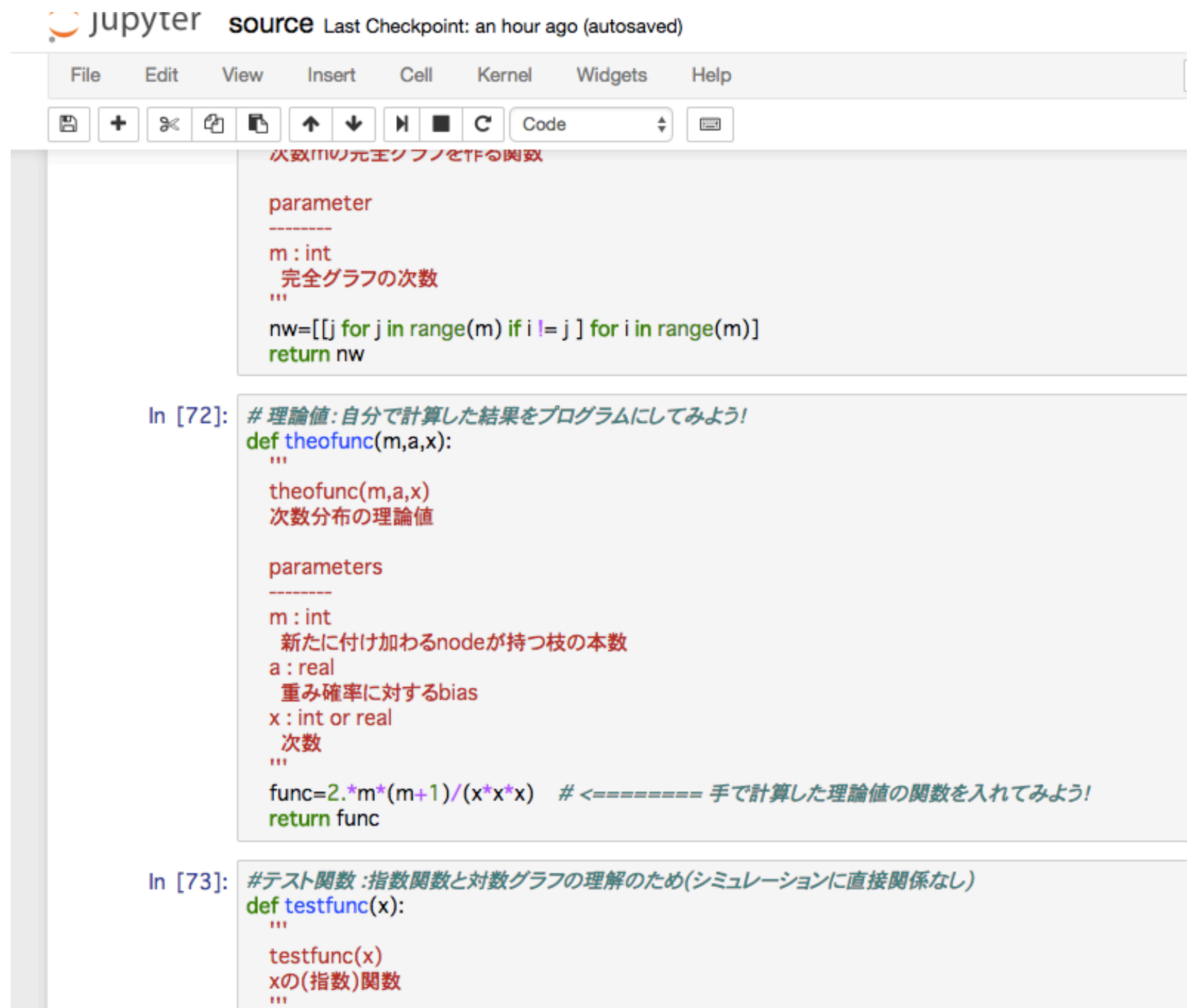
返回值: $y=f(x)$ の y

とおっておけば良い

細かい関数の内容は

プログラムの中の

コメントを見て理解する



```
jupyter source Last Checkpoint: an hour ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
[Icons] Code

# 次数mの完全グラフを作る関数
parameter
-----
m : int
    完全グラフの次数
'''
nw=[[j for j in range(m) if i != j ] for i in range(m)]
return nw

In [72]: # 理論値: 自分で計算した結果をプログラムにしてみよう!
def theofunc(m,a,x):
'''
    theofunc(m,a,x)
    次数分布の理論値

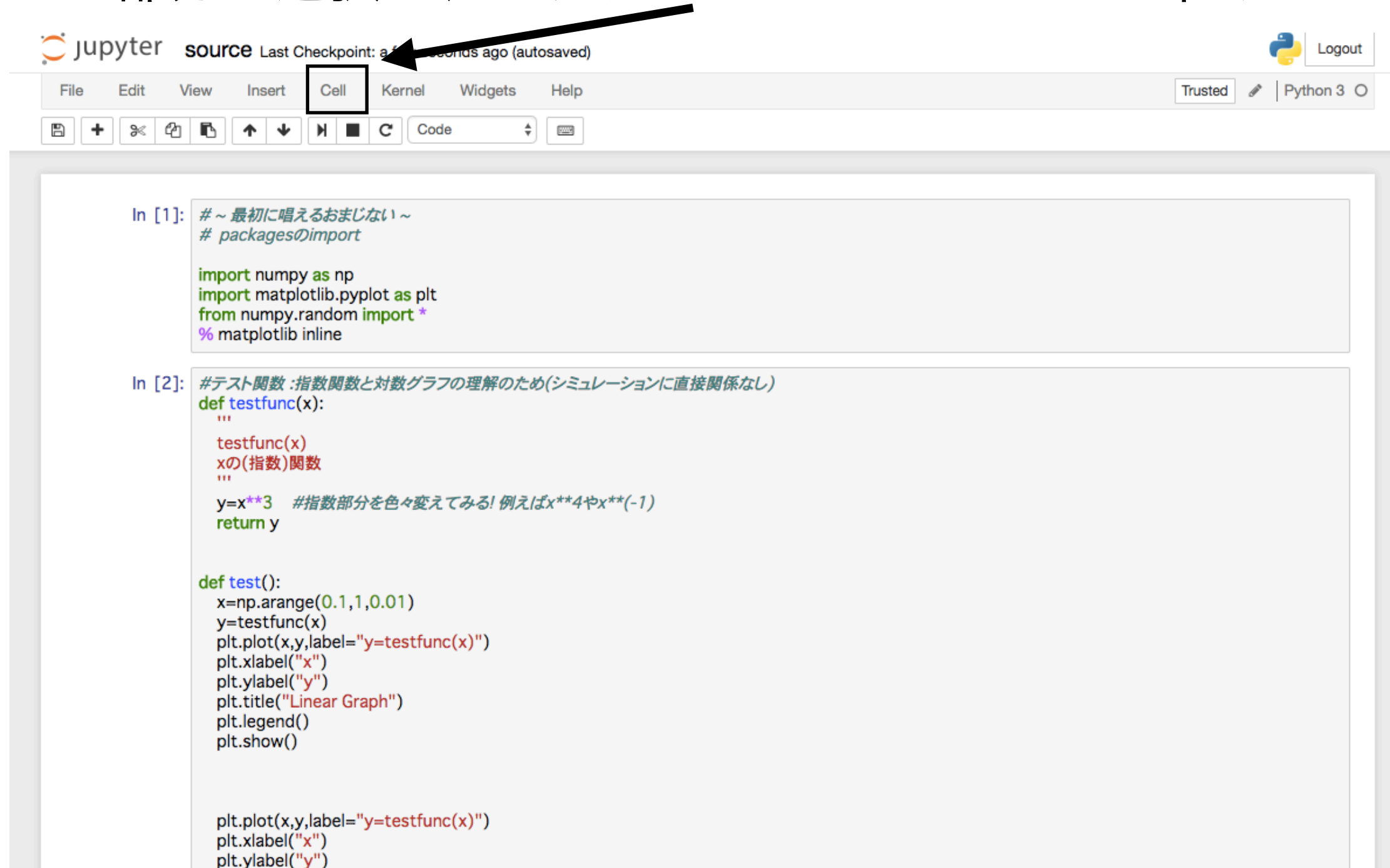
    parameters
    -----
    m : int
        新たに付け加わるnodeが持つ枝の本数
    a : real
        重み確率に対するbias
    x : int or real
        次数
'''
func=2.*m*(m+1)/(x*x*x) # <===== 手で計算した理論値の関数を入れてみよう!
return func

In [73]: #テスト関数 : 指数関数と対数グラフの理解のため(シミュレーションに直接関係なし)
def testfunc(x):
'''
    testfunc(x)
    xの(指数)関数
'''
```


プログラムの実行(べき分布の確認)

1. プログラム内の以下の「#テスト関数～」の部分を探す。

2. この部分を選択し、上タブのCell→Run Cellsを叩く



```
In [1]: # ~ 最初に唱えるおまじない ~
# packagesのimport

import numpy as np
import matplotlib.pyplot as plt
from numpy.random import *
% matplotlib inline

In [2]: #テスト関数 :指数関数と対数グラフの理解のため(シミュレーションに直接関係なし)
def testfunc(x):
    """
    testfunc(x)
    xの(指数)関数
    """
    y=x**3 #指数部分を色々変えてみる! 例えばx**4やx**(-1)
    return y

def test():
    x=np.arange(0.1,1,0.01)
    y=testfunc(x)
    plt.plot(x,y,label="y=testfunc(x)")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Linear Graph")
    plt.legend()
    plt.show()

    plt.plot(x,y,label="y=testfunc(x)")
    plt.xlabel("x")
    plt.ylabel("y")
```

プログラムの実行(ヘッダー)

1. プログラム内の以下の「#～最初に唱えるおまじない～」の部分を探す。
2. この部分を選択し、上タブのCell→Run Cellsを叩く

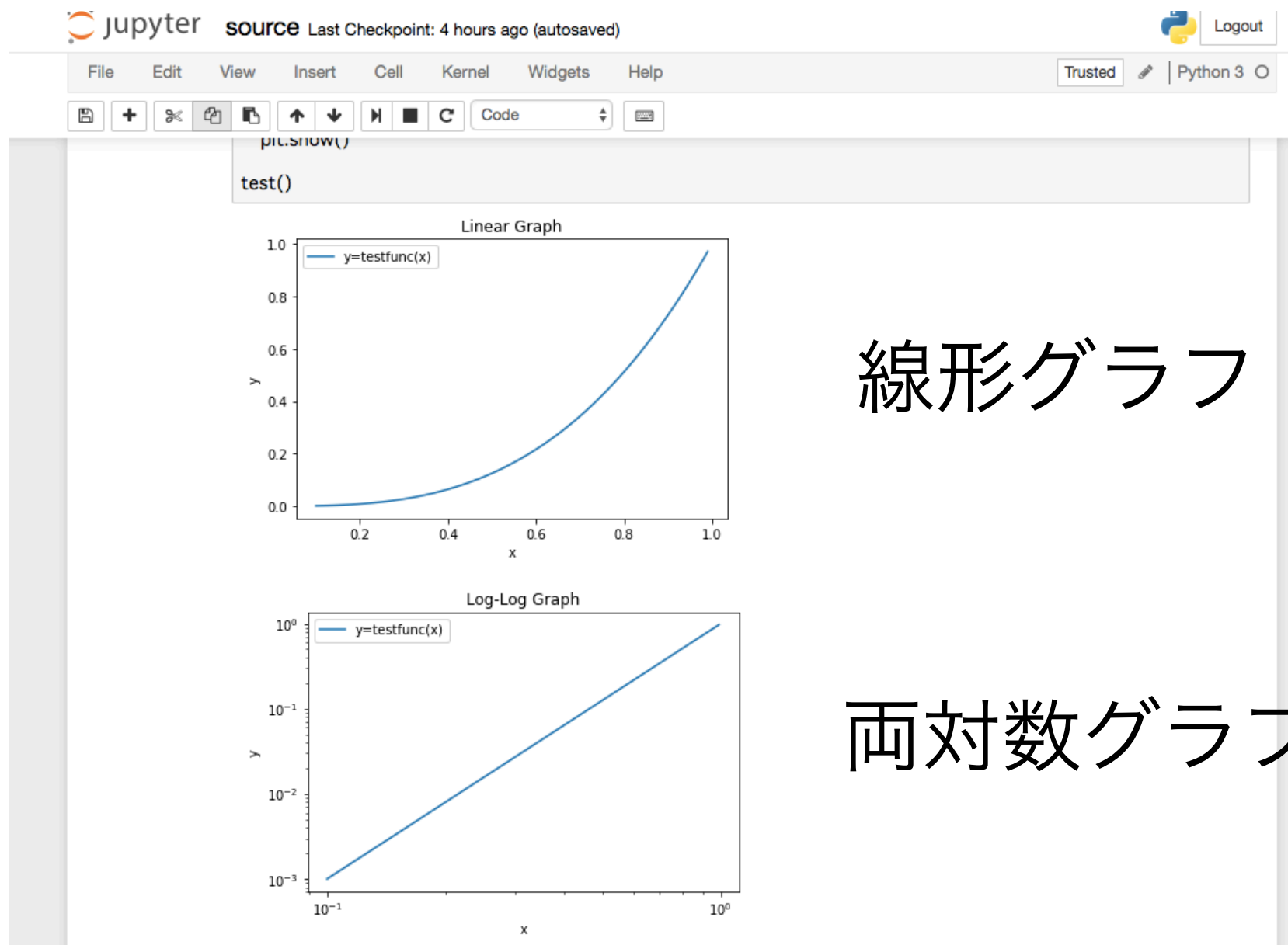
この作業は1回だけ
今後はやらなくて良い



プログラムの実行(べき分布の確認)

3. 実行すると下に図が出力される。(線形グラフと両対数グラフ)

デフォルトは $P(k) = k^3 (y = x^3)$



線形グラフ

両対数グラフ(傾きが指数部分=3)

プログラムを変更してみる

testfunc(x)の中身を変えてみる※例えば x^{**4} は x の4乗

ココ!



```
jupyter source Last Checkpoint: a few seconds ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [1]: # ~ 最初に唱えるおまじない ~
# packagesのimport

import numpy as np
import matplotlib.pyplot as plt
from numpy.random import *
%matplotlib inline

In [2]: #テスト関数 :指数関数と対数グラフの理解のため(シミュレーションに直接関係なし)
def testfunc(x):
    """
    testfunc(x)
    xの(指数)関数
    """
    y=x**3 #指数部分を色々変えてみる! 例えばx**4やx**(-1)
    return y

def test():
    x=np.arange(0.1,1,0.01)
    y=testfunc(x)
    plt.plot(x,y,label="y=testfunc(x)")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Linear Graph")
    plt.legend()
    plt.show()

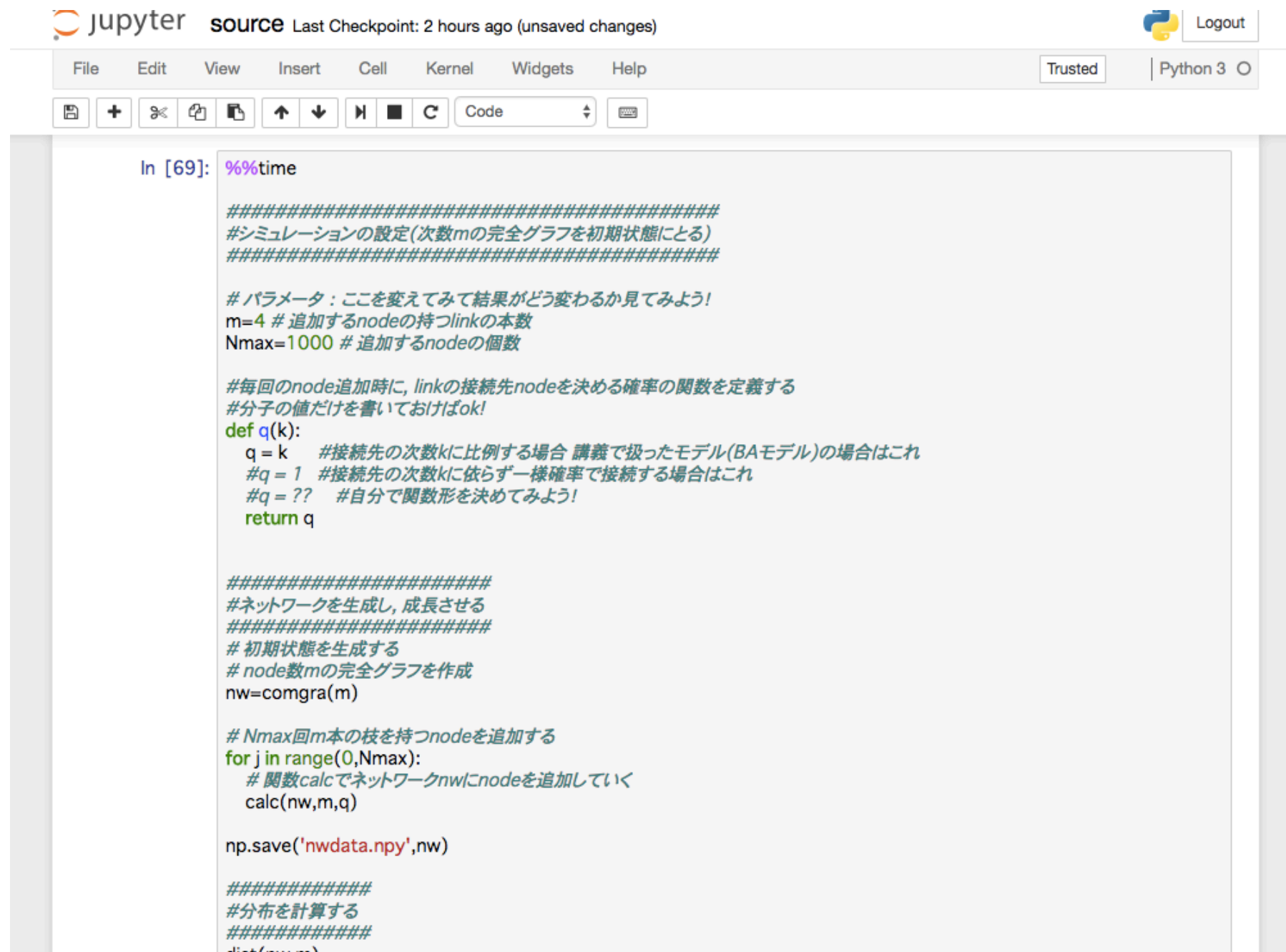
plt.plot(x,y,label="y=testfunc(x)")
plt.xlabel("x")
plt.ylabel("y")
```

できたら先程と同様にCell→Run Cells

プログラムの実行(シミュレーション)

1. プログラムの中で以下の「%%time～」の部分を探す

2. 今度はCell→Run All で実行する



The screenshot shows a Jupyter Notebook interface. At the top, the Jupyter logo and 'source' button are visible, along with a status bar indicating 'Last Checkpoint: 2 hours ago (unsaved changes)'. A 'Logout' button is in the top right. Below the header is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. A 'Trusted' status indicator and 'Python 3' are also present. The main area contains a code cell labeled 'In [69]:'. The code is a simulation script with various comments in Japanese and Python code. It starts with a magic command '%%time' and includes a function 'q(k)' that defines a probability function. The script then generates a network and adds nodes to it.

```
In [69]: %%time

#####
#シミュレーションの設定(次数mの完全グラフを初期状態にとる)
#####

# パラメータ：ここを変えてみて結果がどう変わるか見てみよう!
m=4 # 追加するnodeの持つlinkの本数
Nmax=1000 # 追加するnodeの個数

#毎回のnode追加時に, linkの接続先nodeを決める確率の関数を定義する
#分子の値だけを書いておけばok!
def q(k):
    q = k #接続先の次数kに比例する場合 講義で扱ったモデル(BAモデル)の場合はこれ
    #q = 1 #接続先の次数kに依らず一様確率で接続する場合はこれ
    #q = ?? #自分で関数形を決めてみよう!
    return q

#####
#ネットワークを生成し, 成長させる
#####
# 初期状態を生成する
# node数mの完全グラフを作成
nw=comgra(m)

# Nmax回m本の枝を持つnodeを追加する
for j in range(0,Nmax):
    # 関数calcでネットワークnwにnodeを追加していく
    calc(nw,m,q)

np.save('nwdata.npy',nw)

#####
#分布を計算する
#####
dist(nw,m)
```

プログラムの実行(シミュレーション)

3. 下に次数分布の図が出力される

(青: 理論値、赤: シミュレーション結果)

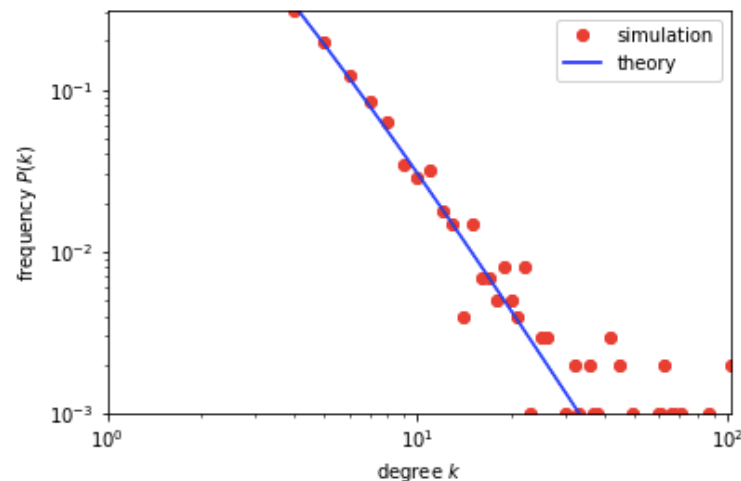
```
jupyter SOURCE Last Checkpoint: 4 hours ago (autosaved) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
# シミュレーション結果
x= np.loadtxt("x.dat") # x.npyには次数の列が入っている
y= np.loadtxt("y.dat") # y.npyには数値シミュレーションした各次数の頻度が入っている

plt.plot(x,y,"ro",label="simulation") # シミュレーション結果のplot

# 理論値 (表示したくない場合は行頭に#を付けば無視されます... `._.`)
y2=np.loadtxt("y2.dat") # y2.npyには理論値の各次数の頻度が入っている
plt.plot(x,y2,"b-",label="theory")

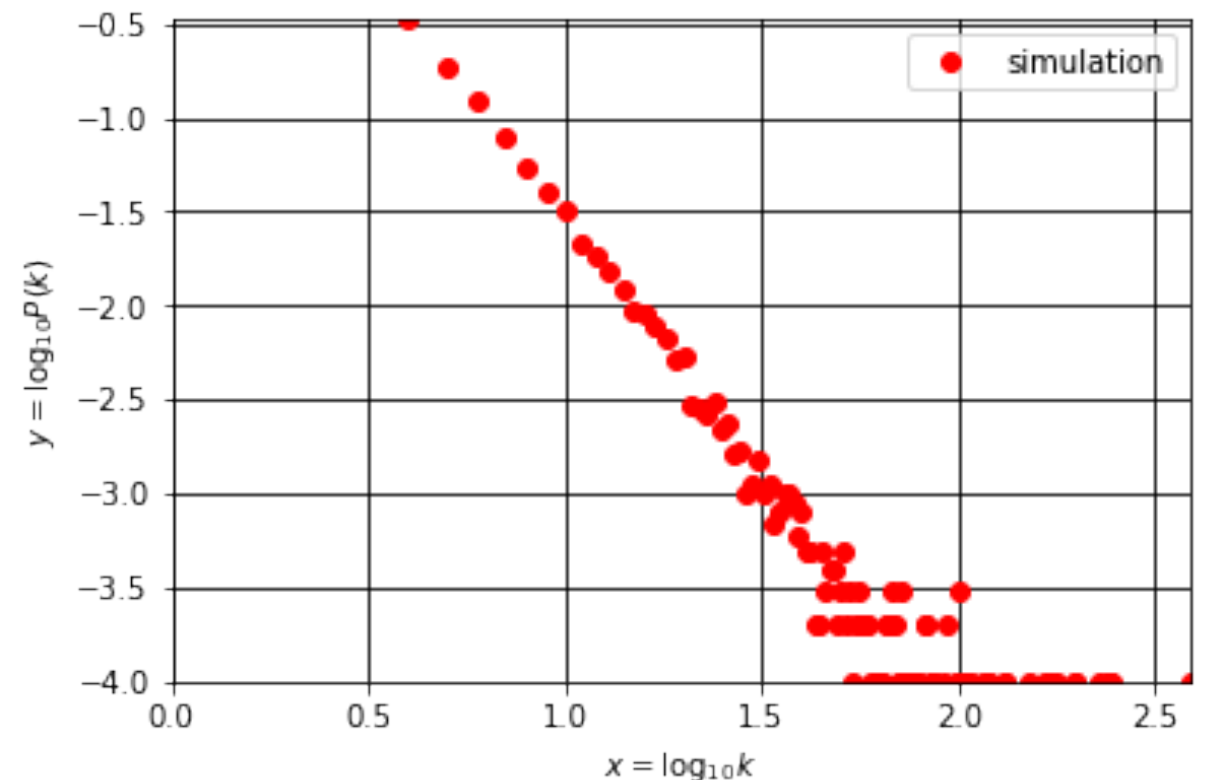
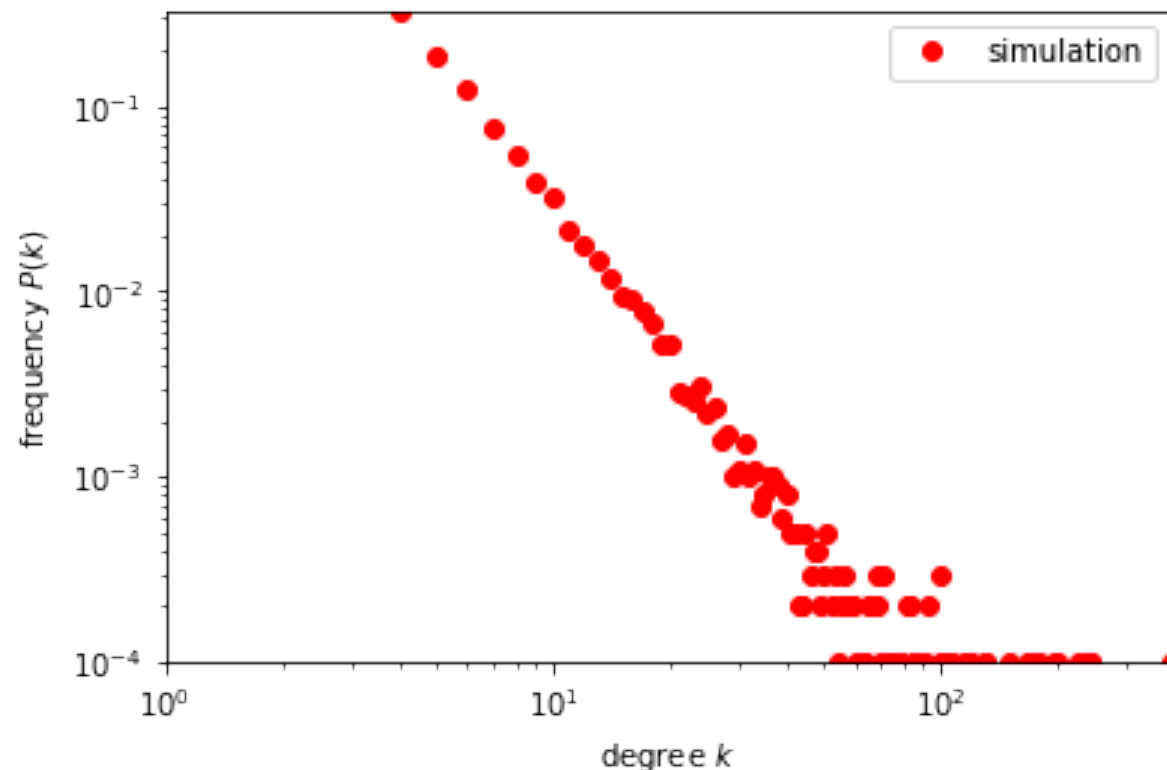
# 表示に関するオプション
plt.xlabel(r"degree $k$")
plt.ylabel(r"frequency $P(k)$")
plt.xlim([1,x[y>0].max()]) # x軸の表示範囲 (1から最大次数まで)
plt.ylim([1/Nmax,y.max()]) # y軸の表示範囲
plt.xscale("log")
plt.yscale("log")
plt.legend()

plt.show() # 表示!!
#plt.savefig("degree_dist.png")
#plt.clf()
```



シミュレーションプログラムの実行結果について 次数分布のグラフを見てみよう

- 下の方のcellに、次数分布のシミュレーション結果が出ている。
(2つのグラフは目盛りの表示方法が異なるだけ)
- このデータからグラフの傾きを求める方法はあるか？

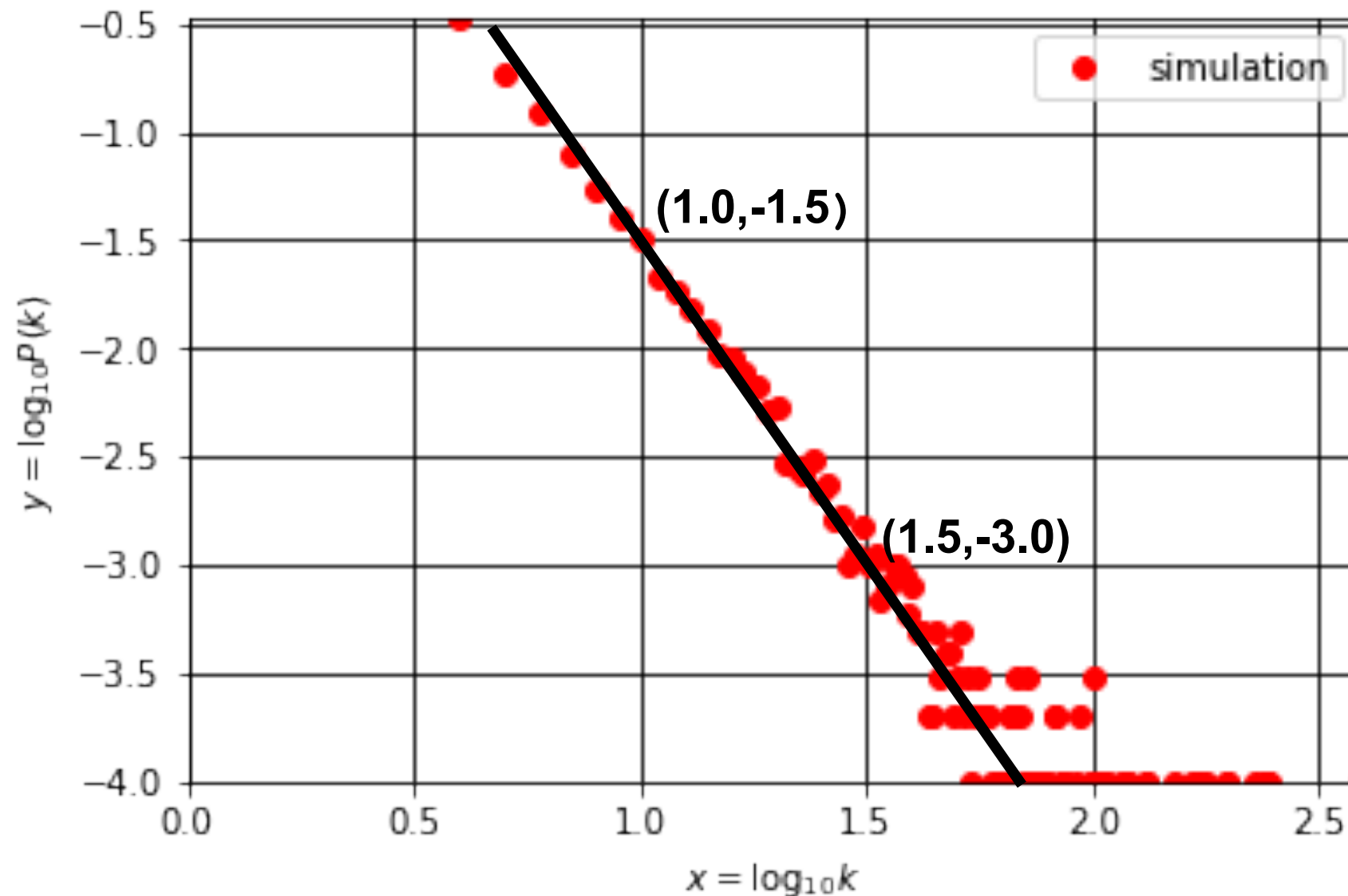


グラフの傾きを計算しよう

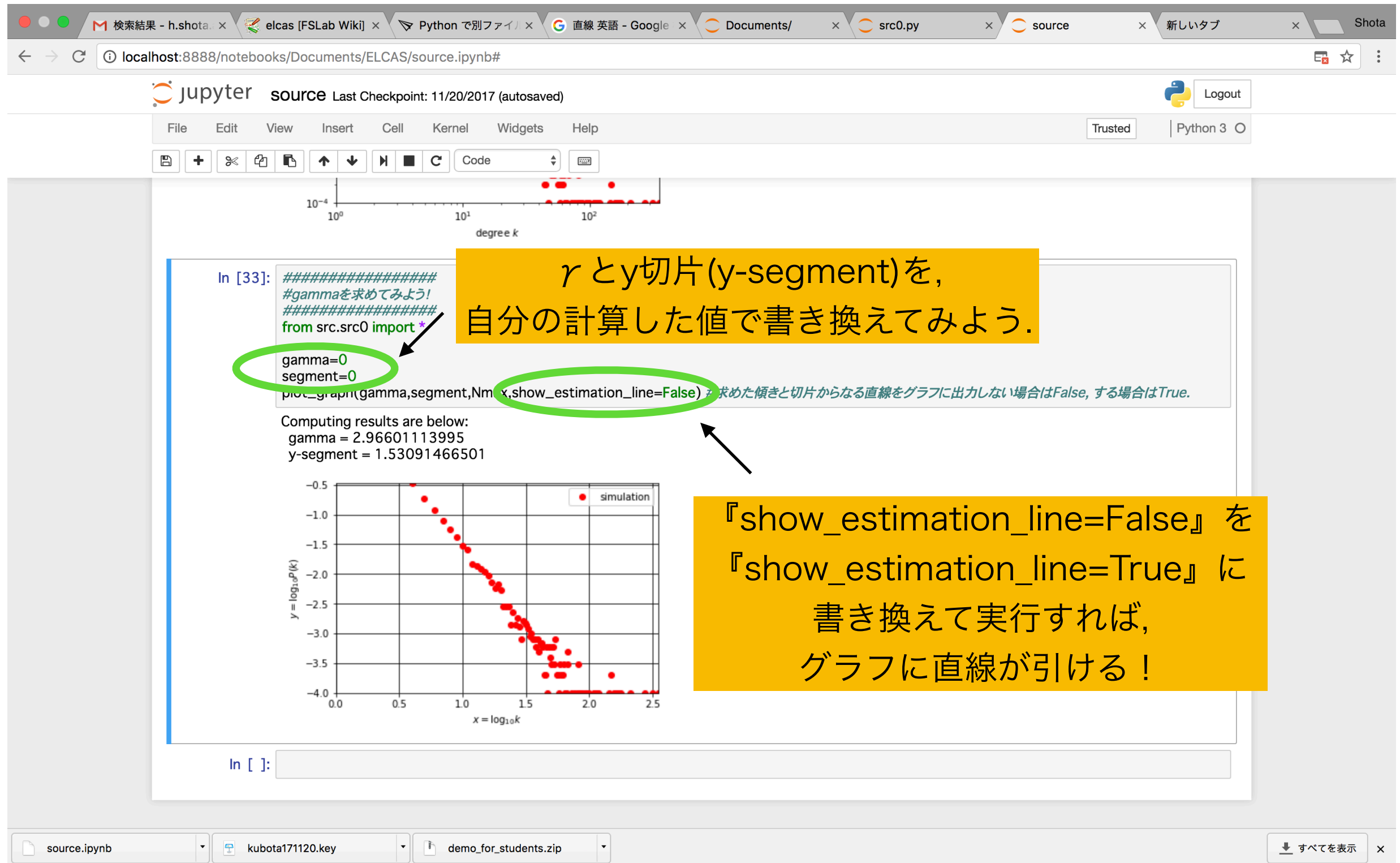
傾き = (yの変化量)/(xの変化量)

$$= (-3.0 + 1.5) / (1.5 - 1.0) = -3.0 = -\gamma$$

y 切片 = 1.5



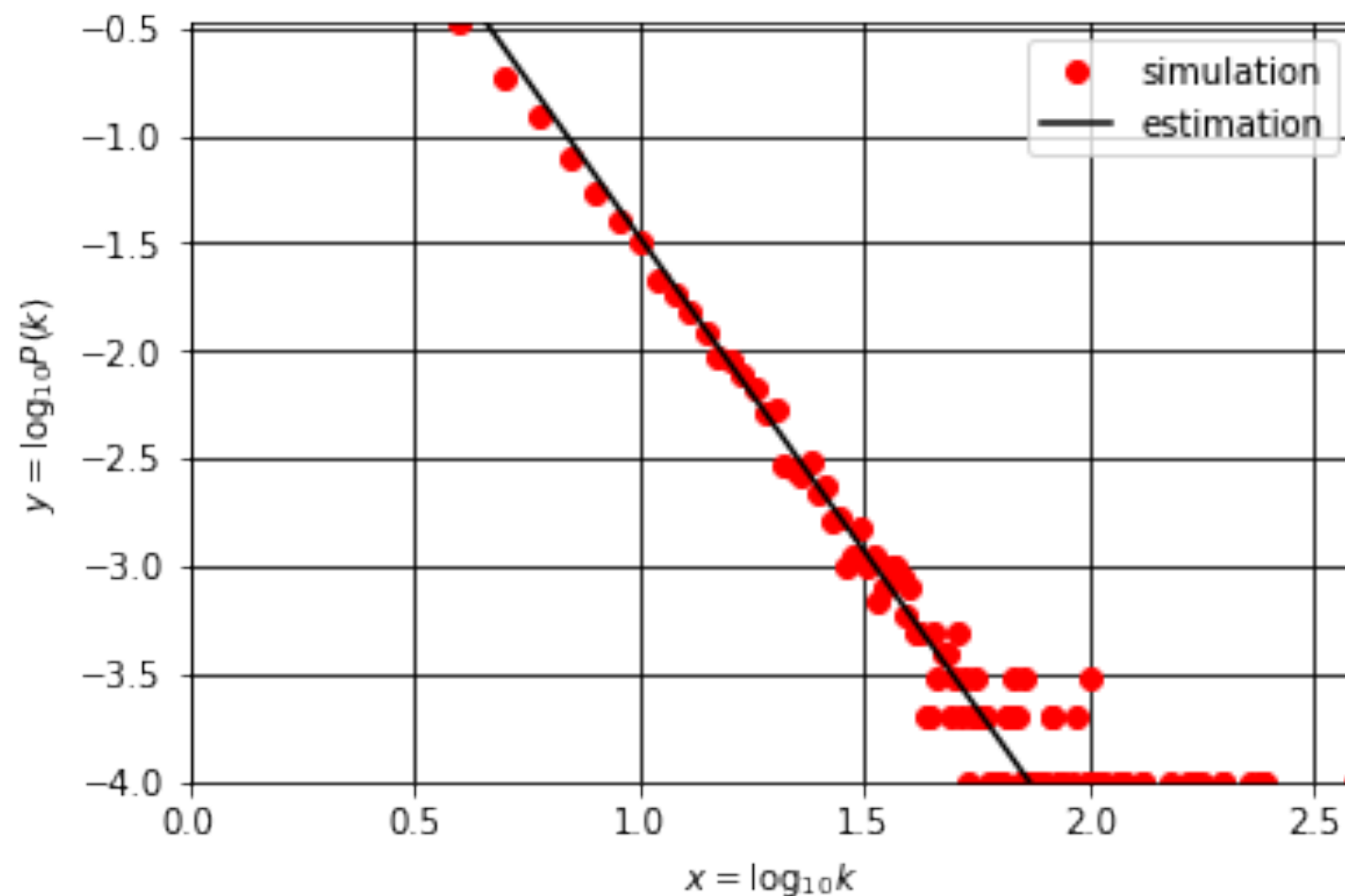
求めた傾きと切片がデータに適合するかを見る。
(グラフの上に, 求めた直線を引いてみる)



※実行は「Ctrl+Enter」でできる

求めた傾きと切片がデータに適合するかを見る。
(グラフの上に, 求めた直線を引いてみる)

このように, およそデータ点に一致する直線が引けるはず！



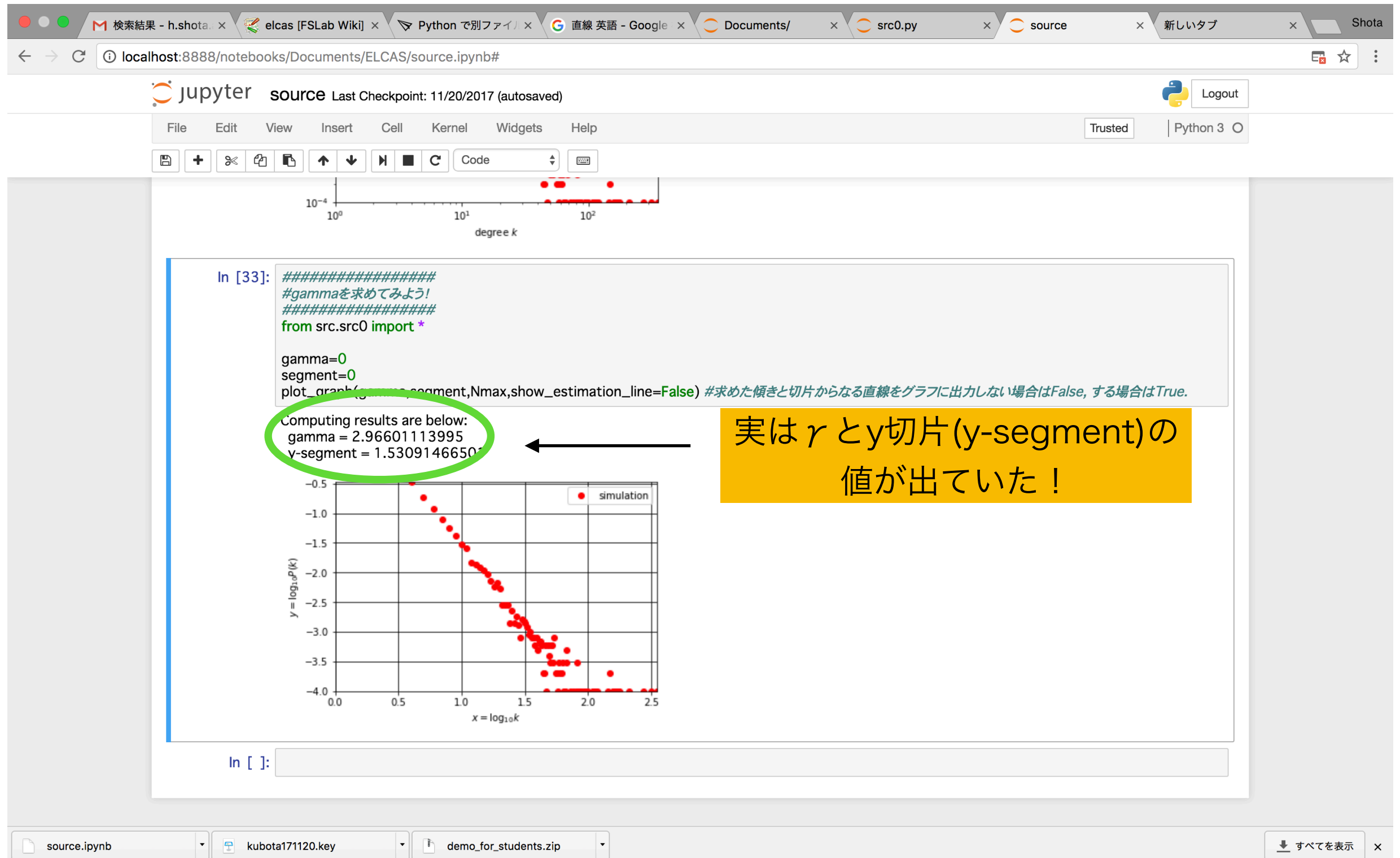
この傾き計算の方法は 万能ではない

- 目分量なので、人によって直線の引き方は異なる！
- 次数(degree) k が大きいと、有限サイズ効果が出ている。

簡単には線が引けない！

大学で学ぶ方法を使って
数値計算的に最適な傾きを導ける。
その方法で導いた値を見てみよう！

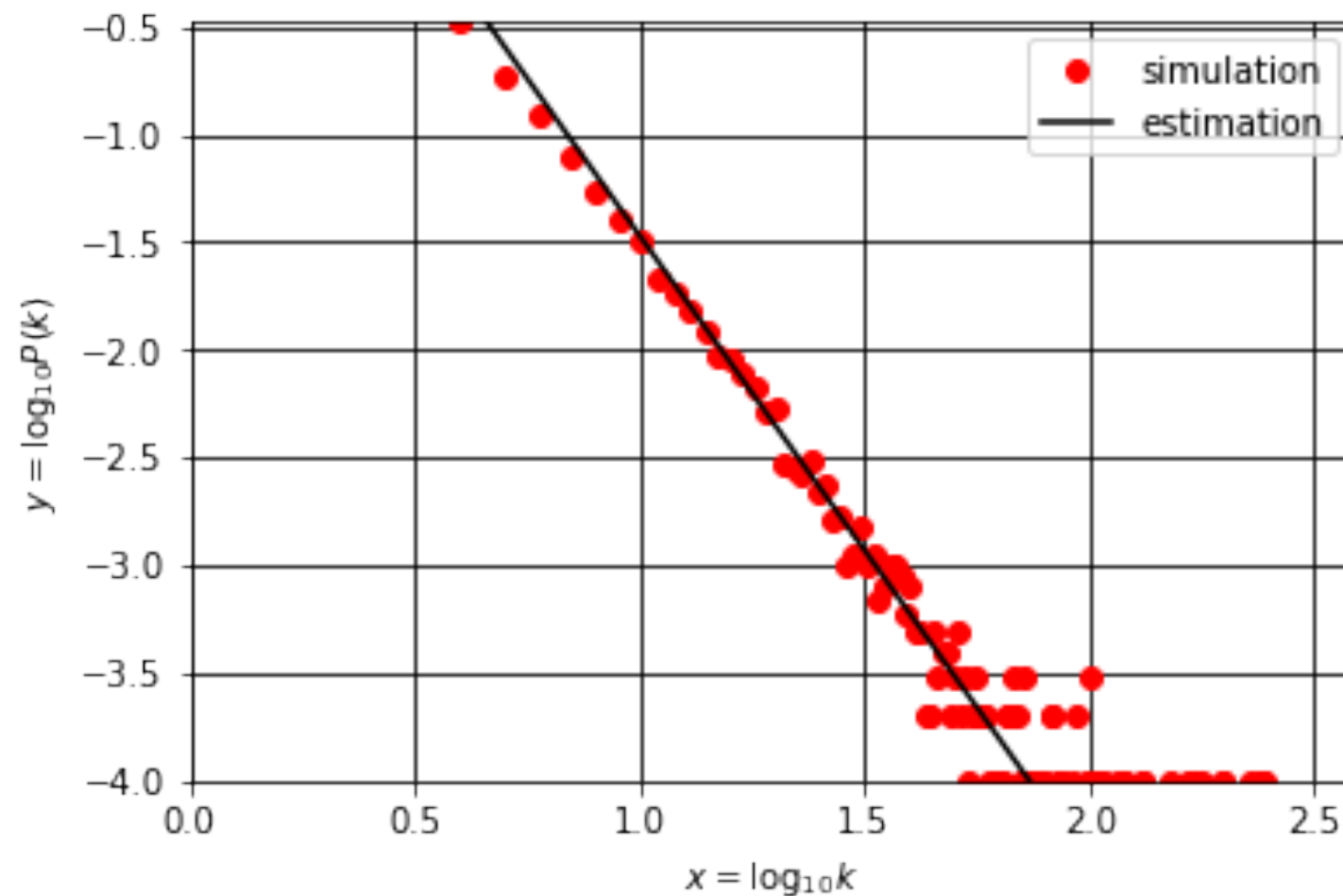
数値計算で導いた傾きを見てみよう、 その値でグラフに直線を引こう



上記の値をプログラムに入力し、もう一度実行してみよう。 ※実行は「Ctrl+Enter」でできる

数値計算で導いた傾きを見てみよう,
その値でグラフに直線を引こう

このように, データ点にフィットするような直線が書けるはず.



P(k)がべき分布の場合, 以上のようにして,
シミュレーションしたグラフが持つ次数分布P(k)についての解析が可能.

ネットワーク描画に関するチュートリアル
(別途IPython Notebookファイルを参照)

プログラムを変更してみる

初期設定、パラメタ設定

ココ! →

```

In [69]: %%time

#####
#シミュレーションの設定(次数mの完全グラフを初期状態にとる)
#####

# パラメータ：ここを変えてみて結果がどう変わるか見てみよう!
m=4 # 追加するnodeの持つlinkの本数
Nmax=1000 # 追加するnodeの個数

#毎回のnode追加時に, linkの接続先nodeを決める確率の関数を定義する
#分子の値だけを書いておけばok!
def q(k):
    q = k #接続先の次数kに比例する場合 講義で扱ったモデル(BAモデル)の場合はこれ
    #q = 1 #接続先の次数kに依らず一様確率で接続する場合はこれ
    #q = ?? #自分で関数形を決めてみよう!
    return q

#####
#ネットワークを生成し, 成長させる
#####
# 初期状態を生成する
# node数mの完全グラフを作成
nw=comgra(m)

# Nmax回m本の枝を持つnodeを追加する
for j in range(0,Nmax):
    # 関数calcでネットワークnwにnodeを追加していく
    calc(nw,m,q)

np.save('nwdata.npy',nw)

#####
#分布を計算する
#####
dist(nw,m)
```

※Nmaxが大きすぎると時間がかかります

接続確率を変更してみる

接続確率 $q(k)$ を変更してみる

ココ！



```
jupyter source Last Checkpoint: 2 hours ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [69]: %%time
#####
#シミュレーションの設定(次数mの完全グラフを初期状態にとる)
#####

# パラメータ：ここを変えてみて結果がどう変わるか見てみよう!
m=4 # 追加するnodeの持つlinkの本数
Nmax=1000 # 追加するnodeの個数

#毎回のnode追加時に, linkの接続先nodeを決める確率の関数を定義する
#分子の値だけを書いておけばok!
def q(k):
    q = k #接続先の次数kに比例する場合 講義で扱ったモデル(BAモデル)の場合はこれ
    #q = 1 #接続先の次数kに依らず一様確率で接続する場合はこれ
    #q = ?? #自分で関数形を決めてみよう!
    return q

#####
#ネットワークを生成し, 成長させる
#####
# 初期状態を生成する
# node数mの完全グラフを作成
nw=comgra(m)

# Nmax回m本の枝を持つnodeを追加する
for j in range(0,Nmax):
    # 関数calcでネットワークnwにnodeを追加していく
    calc(nw,m,q)

np.save('nwdata.npy',nw)

#####
#分布を計算する
#####
dist(nw,m)
```

※ k に依存する部分だけ変えれば良い
例を見て色々変えてみる

グラフプロットに関する説明

接続確率 $q(k)$ を変える時, 次数の頻度分布 $P(k)$ のグラフは
両対数グラフ以外が適することもある.

→様々なタイプのグラフを描くプログラムを用意しておきました

```
In [24]: #####
#上のcellの出力のグラフが直線でない場合, 次数分布P(k)がべき分布でないことが原因と考えられる.
#次数分布P(k)が他の形の場合にも, 見やすくなる形でグラフをplotしてみよう!
#####
from src_elcas.src0h import * #これはおまじないです.

# シミュレーション結果
x= np.loadtxt("x.dat")
y= np.loadtxt("y.dat")

#グラフのプロット
graph_style="normal" # 通常のグラフです.
#graph_style="ylog" # y軸がlogとなる片対数グラフです.
#graph_style="xlog-ylog" # 両軸がlog となる両対数グラフです.
#graph_style="xlog-yloglog" # x軸はlog, y軸はY=log(-(log(y)))です.
#plot_simulated_data(x,y,graph_style)
```

下から2番目のcellに
こんなやつがあるはず.



例えば, 片対数グラフを描きたい場合は, 次のように変えて実行.

```
In [24]: #####
#上のcellの出力のグラフが直線でない場合, 次数分布P(k)がべき分布でないことが原因と考えられる.
#次数分布P(k)が他の形の場合にも, 見やすくなる形でグラフをplotしてみよう!
#####
from src_elcas.src0h import * #これはおまじないです.

# シミュレーション結果
x= np.loadtxt("x.dat")
y= np.loadtxt("y.dat")

#グラフのプロット
#graph_style="normal" # 通常のグラフです.
graph_style="ylog" # y軸がlogとなる片対数グラフです.
#graph_style="xlog-ylog" # 両軸がlog となる両対数グラフです.
#graph_style="xlog-yloglog" # x軸はlog, y軸はY=log(-(log(y)))です.
plot_simulated_data(x,y,graph_style)
```

描きたいスタイル以外に#をつける
(コメントアウト扱いする)

図をプロットする場合は, #を消す
(コメントアウト扱いをやめる)

グラフプロットに関する説明

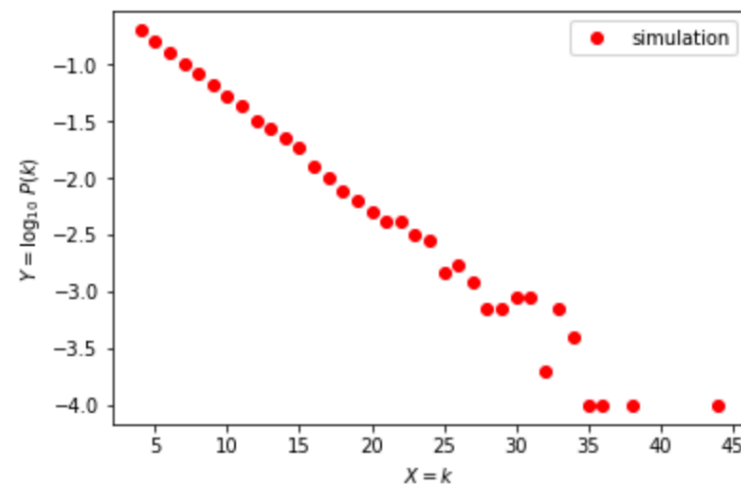
例えば, 接続確率が $q(k)=\text{定数}$ の時, 次数の頻度分布 $P(k)$ のグラフは
片対数グラフが適する.

→ $N=10000$, $q(k)=1$ の時に, このcellの実行結果は次のグラフになる!

```
In [33]: #####
#上のcellの出力のグラフが直線でない場合, 次数分布 $P(k)$ がべき分布でないことが原因と考えられる.
#次数分布 $P(k)$ が他の形の場合にも, 見やすくなる形でグラフをplotしてみよう!
#####
from src_elcas.src0h import * #これはおまじないです.

#シミュレーション結果
x= np.loadtxt("x.dat")
y= np.loadtxt("y.dat")

#グラフのプロット
#graph_style="normal" #通常のグラフです.
graph_style="ylog" #y軸がlogとなる片対数グラフです.
#graph_style="xlog-ylog" #両軸がlogとなる両対数グラフです.
#graph_style="xlog-yloglog" #x軸はlog, y軸は $Y=\log(-(\log(y)))$ です.
plot_simulated_data(x,y,graph_style)
```



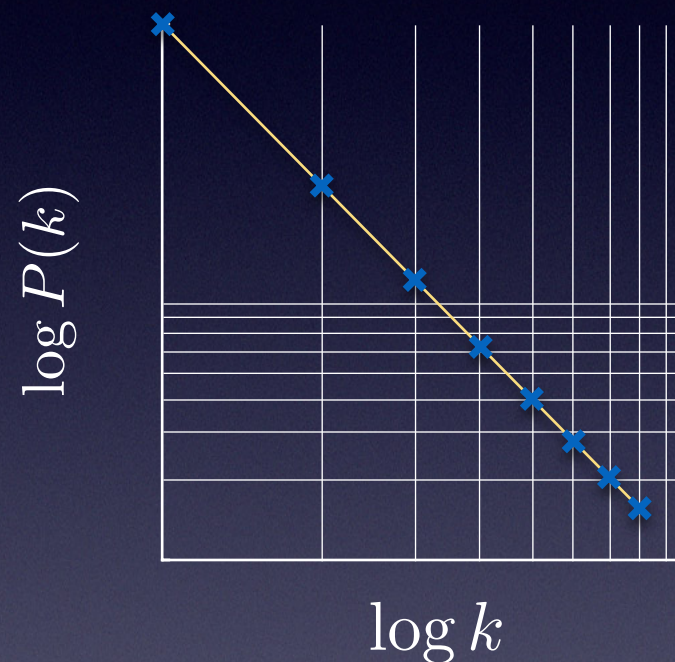
べき分布について

- 📌 べき分布のグラフは両対数をとった軸で見ると便利

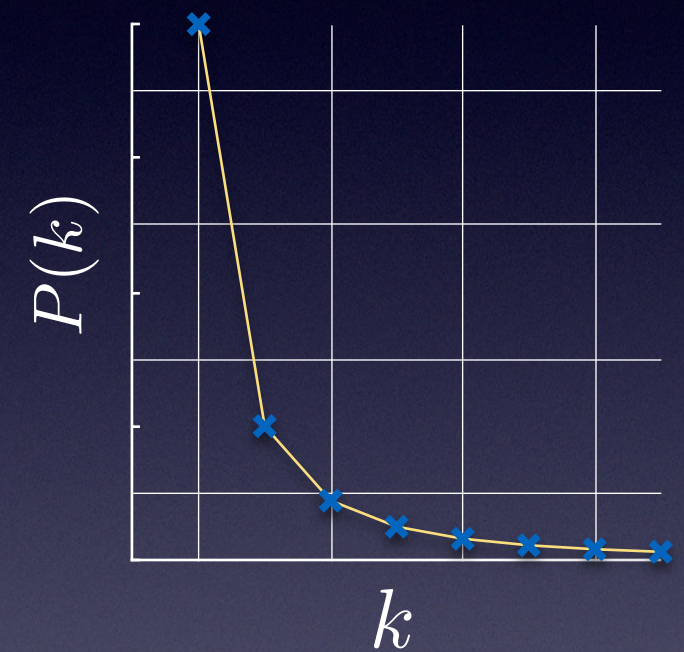
$$P(k) = \frac{A}{k^\gamma}$$

$$\log P(k) = \log A - \gamma \log k$$

両対数プロット → 直線 → べき分布
直線の傾き → γ



両対数グラフ



普通のグラフ

べき分布の判定が容易！

このようなべき分布のネットワークをスケールフリーと呼ぶ！

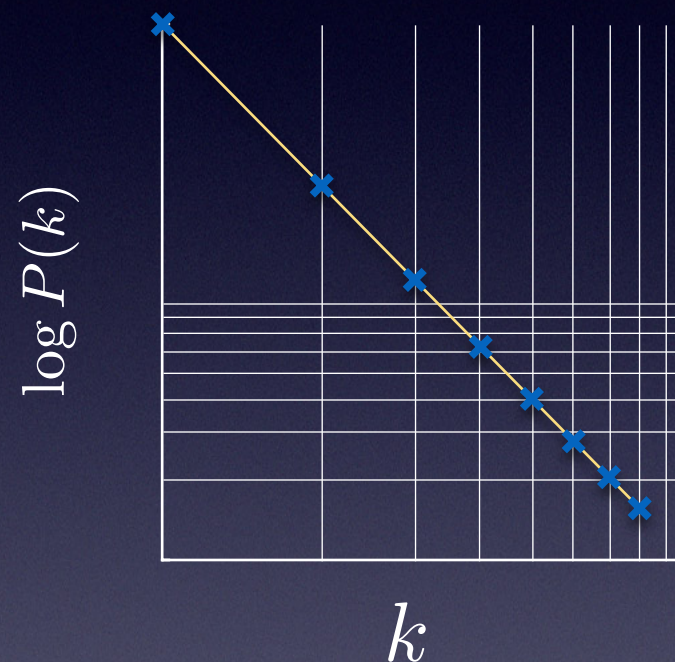
指数分布について

- 指数分布のグラフは片対数をとった軸で見ると便利

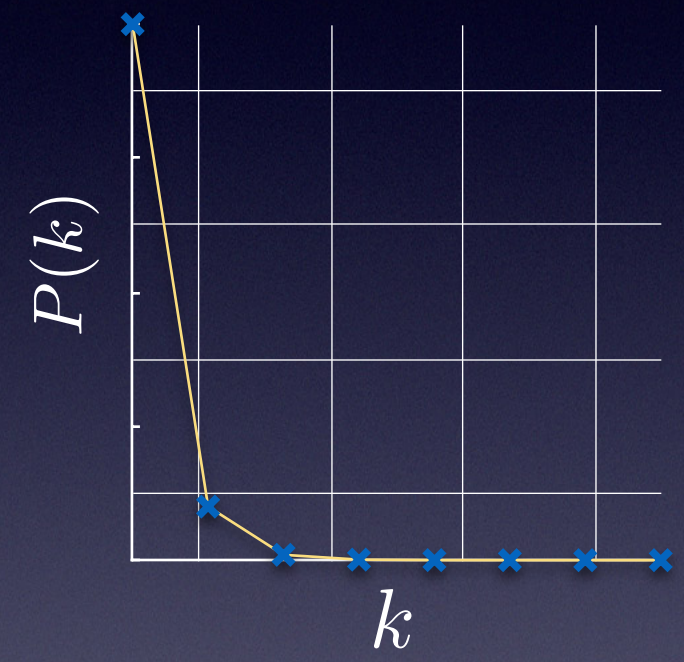
$$P(k) = C \times 10^{-\beta k}$$

$$\log_{10} P(k) = \log_{10} C - \beta k$$

片対数プロット → 直線 → 指数分布
直線の傾き → β



片対数グラフ



普通のグラフ

指数分布の判定が容易！

このような指数分布のネットワークはスケールフリーでない！