In [142…
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sympy import Matrix, init_printing
from scipy.stats import norm
```

First let's set the parametes:

ρ=0.95

σ=0.007

In [143…
```
rho=0.95
sigma=0.007
n=9
```

a)

Now we'll find the upper and lower bound of the grid, by Tauchen's method.

$$\theta_N = m\frac{\sigma}{\sqrt{1-\rho^2}}$$

$$\theta_1 = -m\frac{\sigma}{\sqrt{1-\rho^2}}$$

To be conservative I will use m = 3

In [144…
```
m=3
upp=m*sigma/(1-rho**2)**0.5
low=-upp
```

Now let's generate equidistant points, as well as the grid "separators", compute the cdfs of the limits we have created for our state spaces, then create the Markov chain transition matrix, this is all done with the function built bellow "tau", and the np.space fuction

In [145…
```
def tau(n,rho,sigma,a,b):
    xgrid = np.linspace(low, upp,n )
    def grid(a,b,n):
        w=((np.sqrt(b-a))**2)/(n-1)
        x=np.empty(n-1)
        x[0]=a+w/2
        x[n-2]=b-w/2
        for i in range (n-3):
            x[i+1]=x[i]+w
        return x
    y=grid(a,b,n)
    trm = np.zeros((n, n-1))
    for j in range(n-1):
        for i in range(n):
            trm[i, j] = norm.cdf(y[j],loc=rho*xgrid[i],scale=sigma)
    trmo=np.zeros((n,n))
    for i in range(n):
        trmo[i,0]=trm[i,0]
        trmo[i,n-1]=1-trm[i,n-2]
        for j in range(n-2):
            trmo[i,j+1]=trm[i,j+1]-trm[i,j]
    return trmo
```

```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
```

In [146…
```
def show(X):
    init_printing()
    display(Matrix(np.round(X,5)))
print("Tauchen's method transition matrix:")

show(trmo)
```

```
Tauchen's method transition matrix:
```

$$\begin{bmatrix} 0.76442 & 0.23469 & 0.0009 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.05923 & 0.7405 & 0.19967 & 0.00059 & 0 & 0 & 0 & 0 & 0 \\ 6.0 \cdot 10^{-5} & 0.07471 & 0.7569 & 0.16795 & 0.00039 & 0 & 0 & 0 & 0 \\ 0 & 0.0001 & 0.09314 & 0.76688 & 0.13963 & 0.00025 & 0 & 0 & 0 \\ 0 & 0 & 0.00016 & 0.11473 & 0.77023 & 0.11473 & 0.00016 & 0 & 0 \\ 0 & 0 & 0 & 0.00025 & 0.13963 & 0.76688 & 0.09314 & 0.0001 & 0 \\ 0 & 0 & 0 & 0 & 0.00039 & 0.16795 & 0.7569 & 0.07471 & 6.0 \cdot 10^{-5} \\ 0 & 0 & 0 & 0 & 0 & 0.00059 & 0.19967 & 0.7405 & 0.05923 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0009 & 0.23469 & 0.76442 \end{bmatrix}$$

b)

The rouwenhost method is based on a different grid and matrix using the following logic:

$$\theta_N = \sigma_\theta \sqrt{N-1} \quad \theta_1 = -\theta_N \quad where \quad \sigma_\theta^2 = \frac{\sigma^2}{1-\rho^2}$$

$$P_N = p \begin{bmatrix} P_{N-1} & 0 \\ \mathbf{0}' & 0 \end{bmatrix} + (1-p) \begin{bmatrix} \mathbf{0} & P_{N-1} \\ 0 & \mathbf{0}' \end{bmatrix} + (1-p) \begin{bmatrix} \mathbf{0}' & 0 \\ P_{N-1} & \mathbf{0} \end{bmatrix} + p \begin{bmatrix} 0 & \mathbf{0}' \\ \mathbf{0} & P_{N-1} \end{bmatrix}$$

$$p = \frac{1+\rho}{2}, P_2 = \begin{bmatrix} p & 1-p \\ 1-p & p \end{bmatrix}$$

Now i create another fuction to get this matrix:

In [147…
```
def rouwenhorst(n, rho,sigma):
    p = (1 + rho) / 2
    maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
    mini=-maxi
    if n == 2:
        theta = np.array([[p, 1 - p], [1 - p, p]])

    else :
        p1 = np.zeros((n, n))
        p2 = np.zeros((n, n))
        p3 = np.zeros((n, n))
        p4 = np.zeros((n, n))

        new_mat = rouwenhorst(n - 1, rho, sigma)

        p1[:n - 1, :n - 1] = p * new_mat
        p2[:n - 1, 1:] = (1 - p) * new_mat
        p3[1:, :-1] = (1 - p) * new_mat
        p4[1:, 1:] = p * new_mat

        theta = p1 + p2 + p3 + p4
        for i in range (n):
```

```
            theta[i: ,:] = (theta[i:,:] / sum(theta[i]))

    return theta
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
```

In [148…

```
print("ROU method transition matrix:")

show(trm1)
```

ROU method transition matrix:

$$\begin{bmatrix}
0.81665 & 0.16752 & 0.01503 & 0.00077 & 2.0\cdot10^{-5} & 0 & 0 & 0 & 0 \\
0.02094 & 0.82041 & 0.14687 & 0.01129 & 0.00048 & 1.0\cdot10^{-5} & 0 & 0 & 0 \\
0.00054 & 0.04196 & 0.8231 & 0.12605 & 0.00807 & 0.00028 & 1.0\cdot10^{-5} & 0 & 0 \\
1.0\cdot10^{-5} & 0.00161 & 0.06303 & 0.82472 & 0.10511 & 0.00538 & 0.00014 & 0 & 0 \\
0 & 6.0\cdot10^{-5} & 0.00323 & 0.08409 & 0.82526 & 0.08409 & 0.00323 & 6.0\cdot10^{-5} & 0 \\
0 & 0 & 0.00014 & 0.00538 & 0.10511 & 0.82472 & 0.06303 & 0.00161 & 1.0\cdot10 \\
0 & 0 & 1.0\cdot10^{-5} & 0.00028 & 0.00807 & 0.12605 & 0.8231 & 0.04196 & 0.000 \\
0 & 0 & 0 & 1.0\cdot10^{-5} & 0.00048 & 0.01129 & 0.14687 & 0.82041 & 0.020 \\
0 & 0 & 0 & 0 & 2.0\cdot10^{-5} & 0.00077 & 0.01503 & 0.16752 & 0.816
\end{bmatrix}$$

The matrix's presented above only display the first five digits after the dot, this was made for visual reasons, the trmo and trm1 matrix's that contain the full numbers (and add up to 1 in each row) is what is going to be used for calculations.

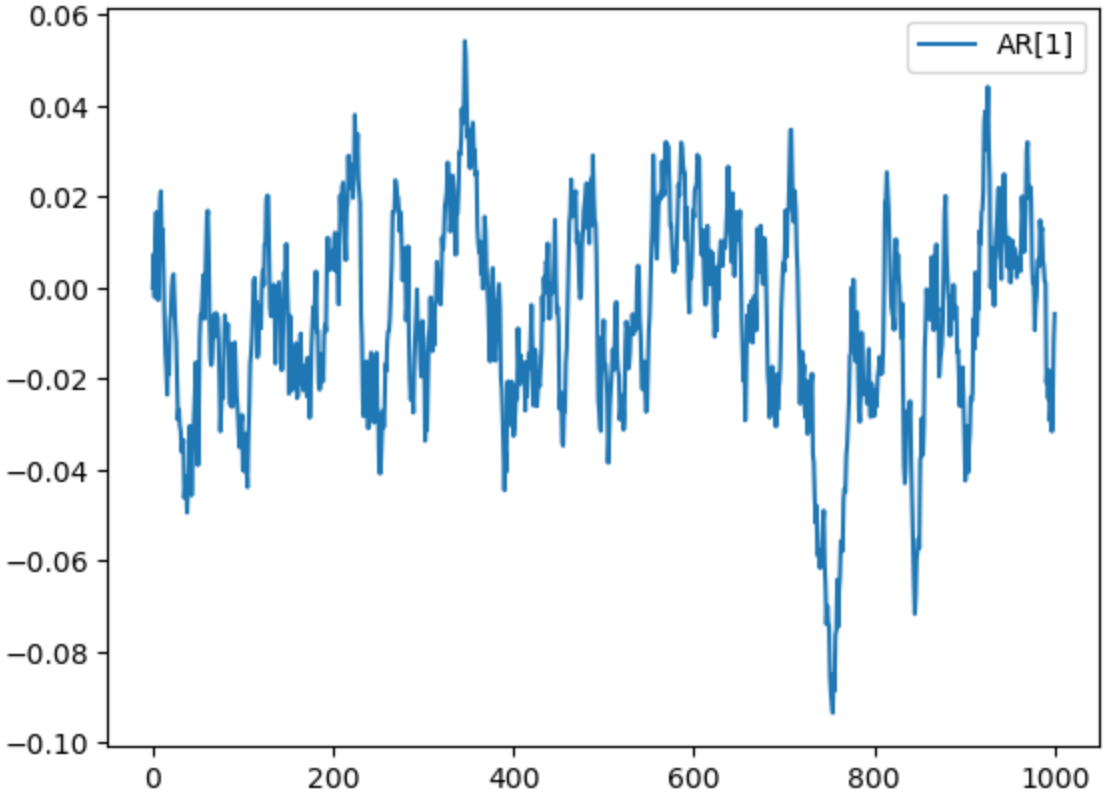
(If interested the amount of numbers after the dot can be changed in the show function)

c)

Now we can simulate the process, starting with the continuous one to get its shocks:

In [149…

```
np.random.seed(1658)
x=np.empty(1001)
x[0]=0
alpha=0.95
T=1000
shocks=np.random.normal(size=1000,scale=0.007)
for t in range(T):
    x[t+1]=x[t]*alpha + shocks[t]
plt.plot(x, label='AR[1]')
plt.legend()
plt.show()
```

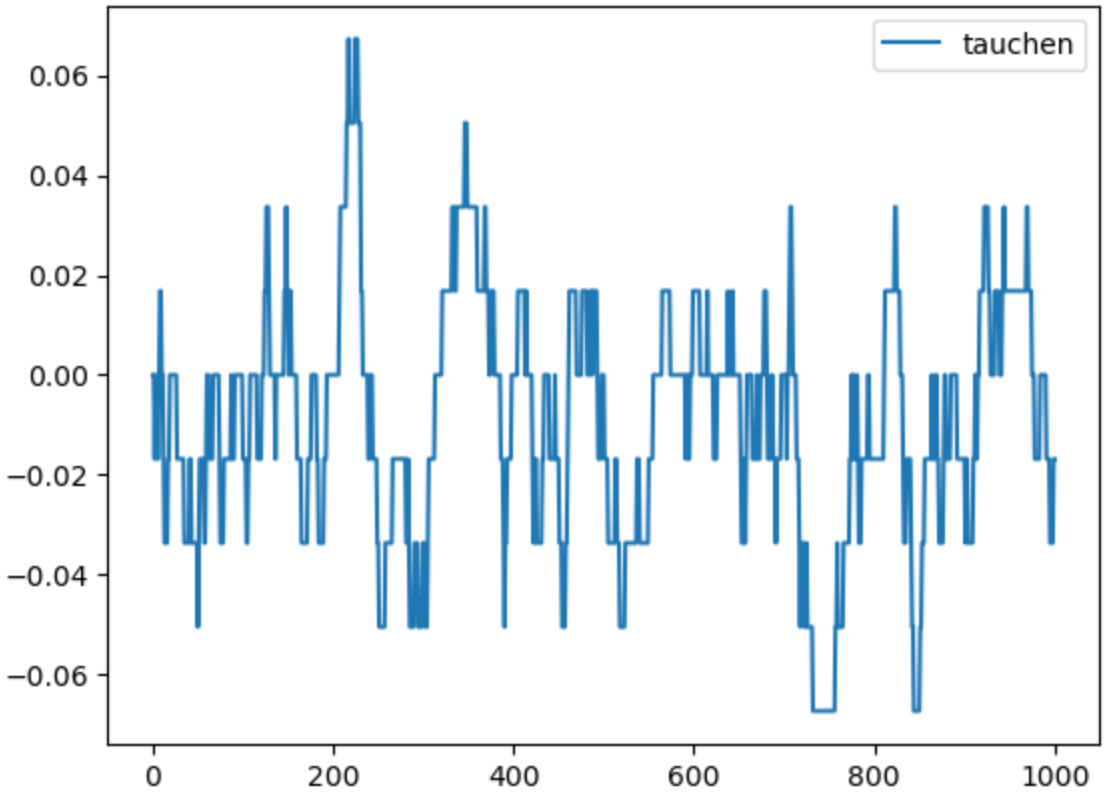


We then get the cdf of the shocks, and choose the next state if the process in the discrete case as the first one that the sum of the transition matrix of the current step (starting from the left)

In [150…

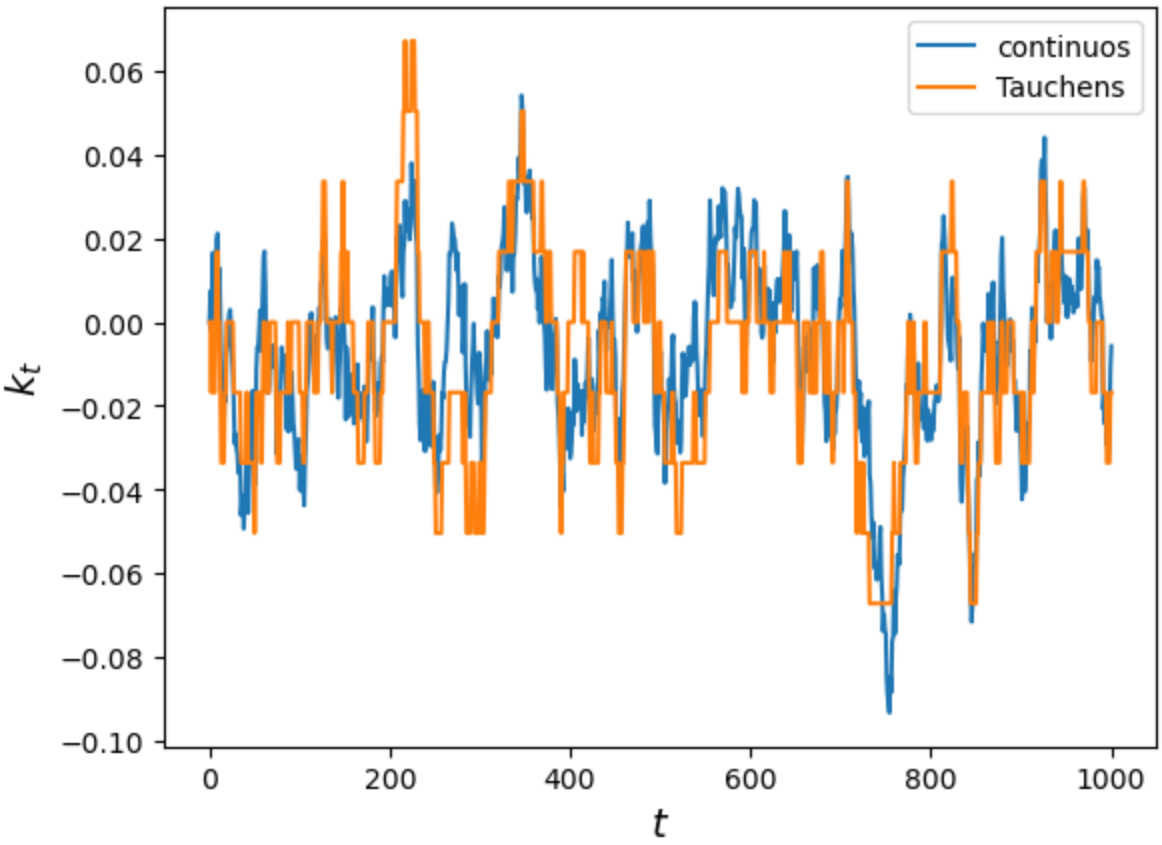
```
shocks_cdf= norm.cdf(shocks,scale=sigma)
nsteps = 1000
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
trm = np.zeros((n, n+1))
trm[:, 1:] = trmo
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trmo[int(current_state[i]), :]), shocks
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])]

plt.plot(simulated_states, label='tauchen')
plt.legend()
plt.show()
```



In [151…

```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states, label='Tauchens')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```

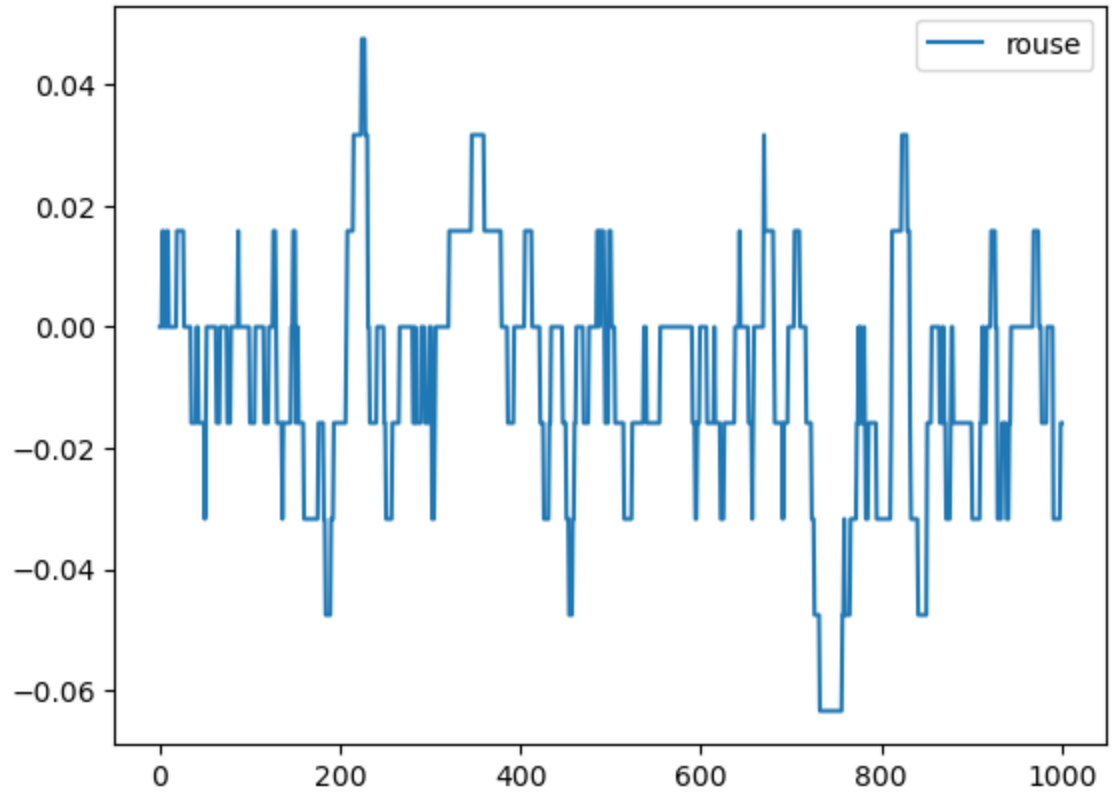


In [153…

```
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
```

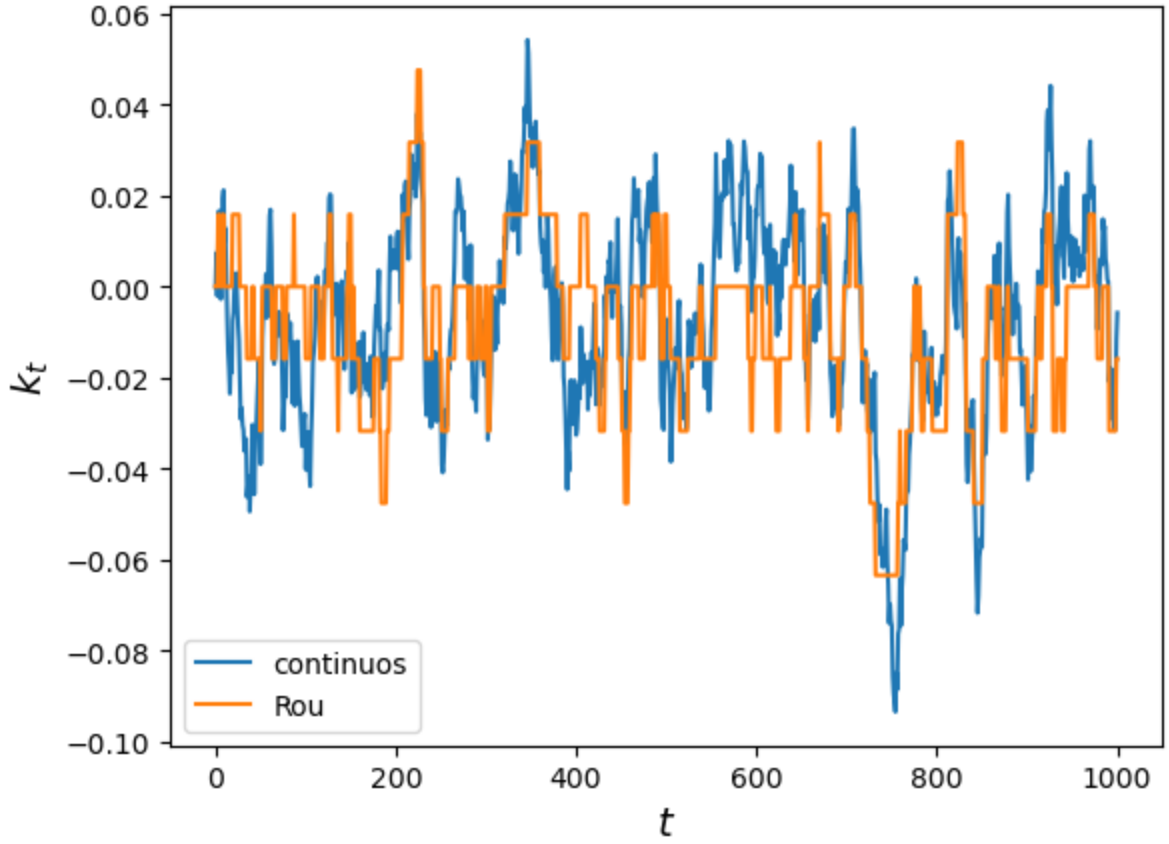
```
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trm1[int(current_state1[i]), :]), shock
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])]

plt.plot(simulated_states1, label='rouse')
plt.legend()
plt.show()
```



In [154…

```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states1, label='Rou')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```

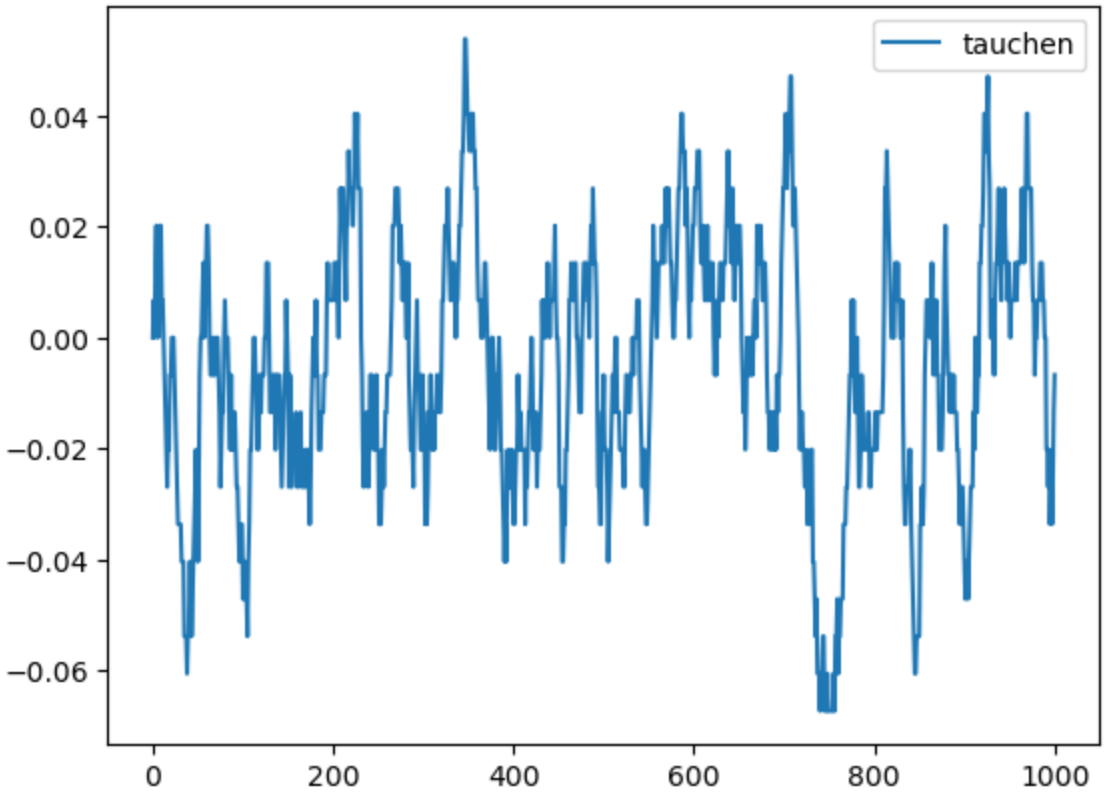


The graphs are kind off close but let's try n=21

In [155…
```
n=21
```

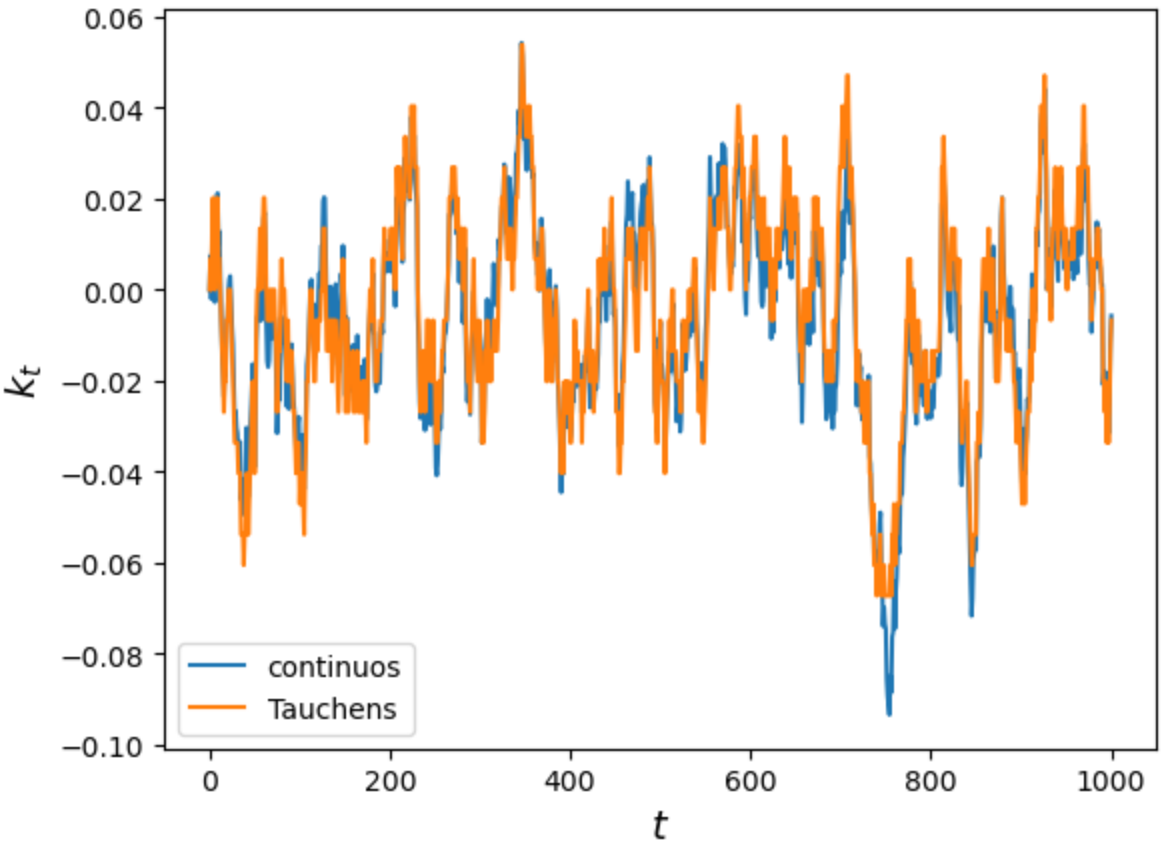
In [156…
```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trmo[int(current_state[i]), :]), shocks
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])]

plt.plot(simulated_states, label='tauchen')
plt.legend()
plt.show()
```



In [157…

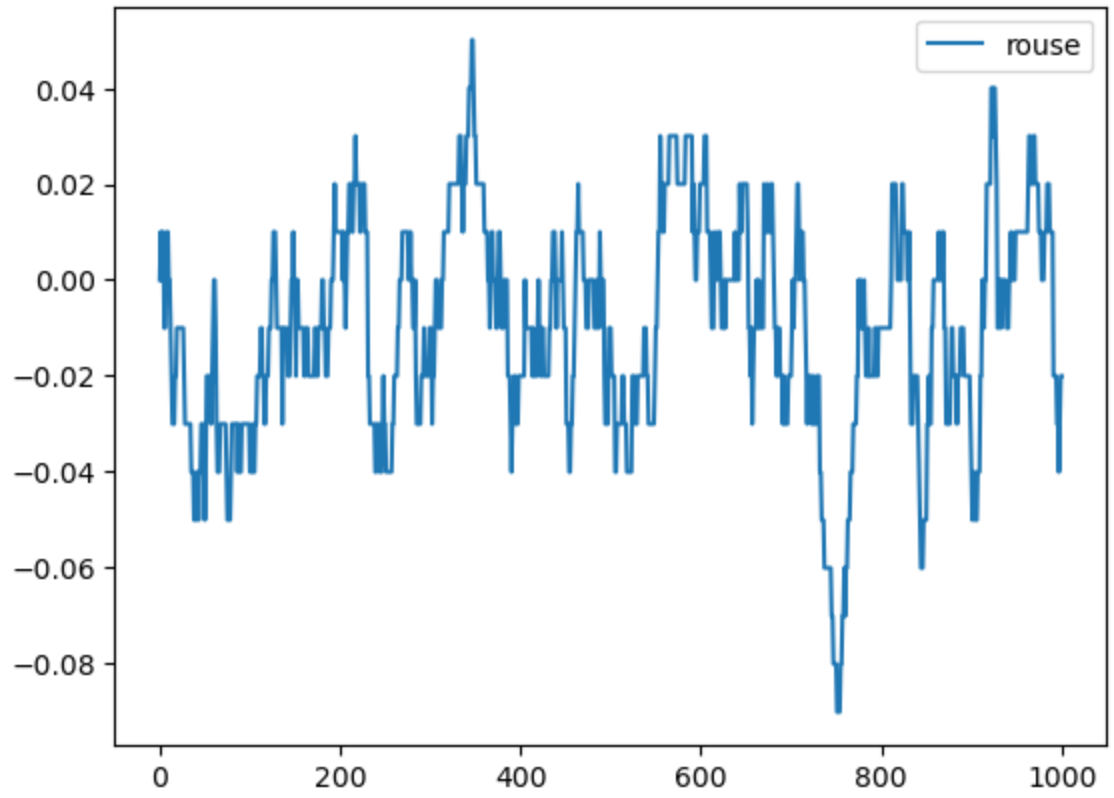
```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states, label='Tauchens')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```



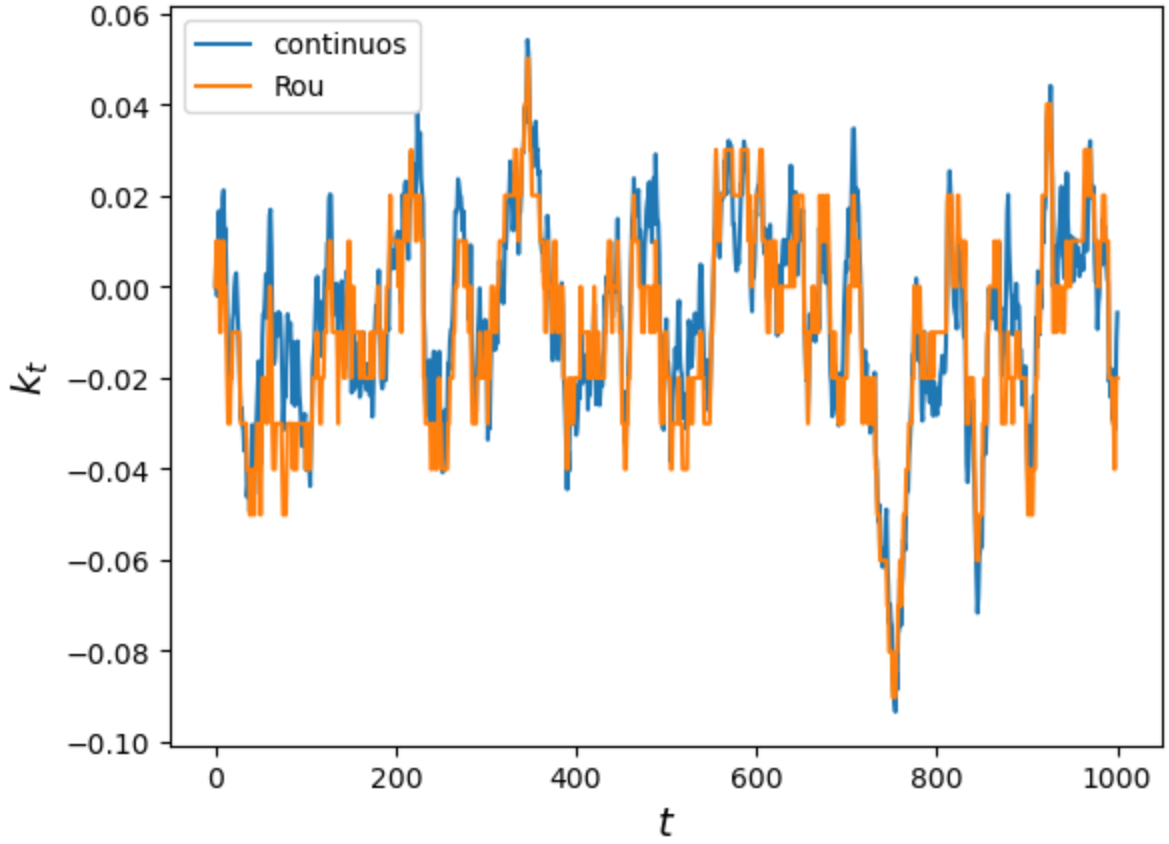
It is clear that we got much closer to the real distribuition, let's see with Rou

In [158…
```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trm1[int(current_state1[i]), :]), shock
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])]

plt.plot(simulated_states1, label='rouse')
plt.legend()
plt.show()
```



In [159…
```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states1, label='Rou')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```



It is also much closer to the "real" distribution

d)

Let's go back to n=9 ang do the regression:

In [160…
```
from sklearn.linear_model import LinearRegression
```

In [161…
```
n=9
```

In [162…
```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trmo[int(current_state[i]), :]), shocks
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])]
model = LinearRegression()
model.fit(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshape(-1, 1))
r_sq = model.score(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshap


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.00051535]
Coefficients: [[0.94094989]]
coefficient of determination: 0.8853571143818355
```

The regression came out pretty acurate, let's checke for row

In [163…
```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0

for i in range(nsteps-1):
    next_state_index = np.searchsorted(np.cumsum(trm1[int(current_state1[i]), :]), shock
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])]
model = LinearRegression()
model.fit(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].reshape(-1, 1
r_sq = model.score(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].resh


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.00057746]
Coefficients: [[0.93400629]]
coefficient of determination: 0.8723677556622678
```

It is still pretty good, but let's try with n=21

In [166…
```
n=21
```

In [167…
```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
for i in range(nsteps-1):
    next_state_index = np.searchsorted(np.cumsum(trmo[int(current_state[i]), :]), shocks
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])]
model = LinearRegression()
model.fit(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshape(-1, 1))
r_sq = model.score(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshap


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.00030133]
Coefficients: [[0.94482163]]
coefficient of determination: 0.8926879115636793
```

In [168…
```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0
for i in range(nsteps-1):
    next_state_index = np.searchsorted(np.cumsum(trm1[int(current_state1[i]), :]), shock
```

```
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])]
model = LinearRegression()
model.fit(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].reshape(-1, 1
r_sq = model.score(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].resh


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.00051436]
Coefficients: [[0.94748716]]
coefficient of determination: 0.8977319228773639
```

They both came out better

e) doing it again for:

ρ=0.99

In [208…
```
rho=0.99
n=9
m=3
upp=m*sigma/(1-rho**2)**0.5
low=-upp
```

In [209…
```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
show(trmo)
```

$$\begin{bmatrix} 0.99277 & 0.00723 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.00242 & 0.99135 & 0.00623 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.00285 & 0.99179 & 0.00536 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.00335 & 0.99206 & 0.00459 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.00393 & 0.99215 & 0.00393 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.00459 & 0.99206 & 0.00335 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.00536 & 0.99179 & 0.00285 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.00623 & 0.99135 & 0.00242 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.00723 & 0.99277 \end{bmatrix}$$

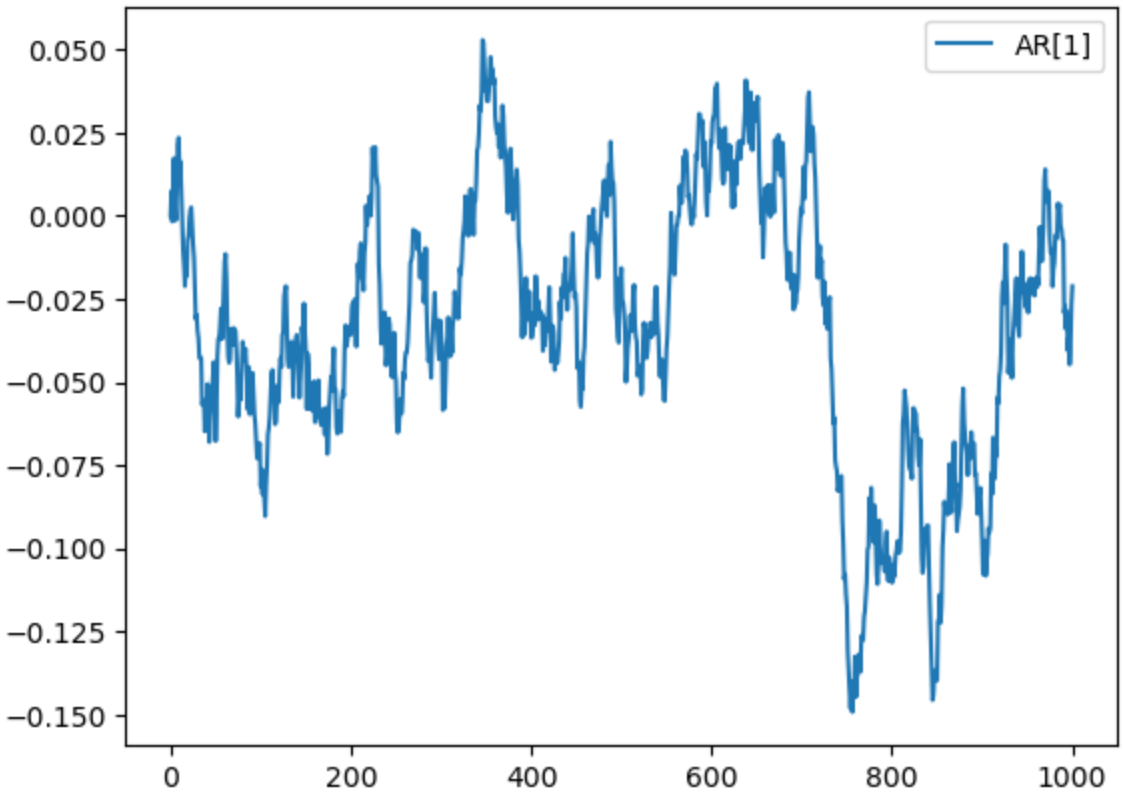
In [210…
```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
show(trm1)
```

$$\begin{bmatrix} 0.96069 & 0.03862 & 0.00068 & 1.0 \cdot 10^{-5} & 0 & 0 & 0 & 0 & 0 \\ 0.00483 & 0.96086 & 0.0338 & 0.00051 & 0 & 0 & 0 & 0 & 0 \\ 2.0 \cdot 10^{-5} & 0.00966 & 0.96098 & 0.02897 & 0.00036 & 0 & 0 & 0 & 0 \\ 0 & 7.0 \cdot 10^{-5} & 0.01448 & 0.96106 & 0.02414 & 0.00024 & 0 & 0 & 0 \\ 0 & 0 & 0.00015 & 0.01931 & 0.96108 & 0.01931 & 0.00015 & 0 & 0 \\ 0 & 0 & 0 & 0.00024 & 0.02414 & 0.96106 & 0.01448 & 7.0 \cdot 10^{-5} & 0 \\ 0 & 0 & 0 & 0 & 0.00036 & 0.02897 & 0.96098 & 0.00966 & 2.0 \cdot 10^{-5} \\ 0 & 0 & 0 & 0 & 0 & 0.00051 & 0.0338 & 0.96086 & 0.00483 \\ 0 & 0 & 0 & 0 & 0 & 1.0 \cdot 10^{-5} & 0.00068 & 0.03862 & 0.96069 \end{bmatrix}$$

Now simulating a new AR[1] process and comparing to the simulations of tauchens and row's

In [211…

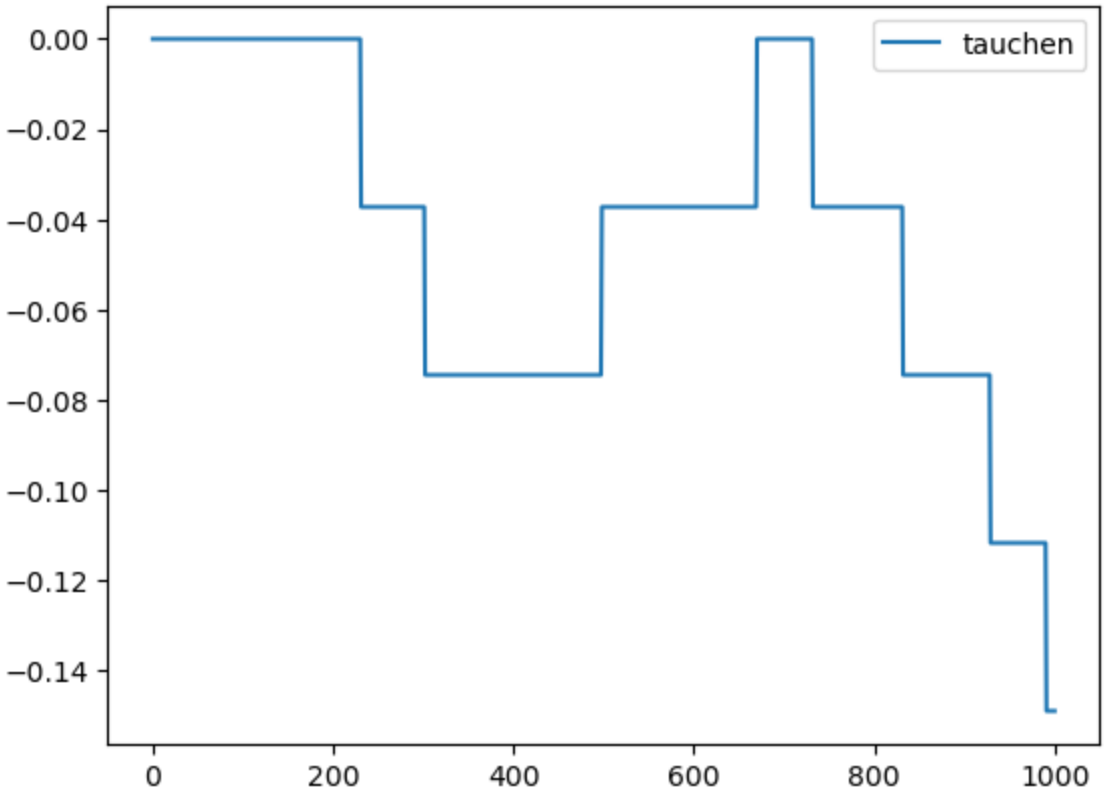
```
np.random.seed(1658)
x=np.empty(1001)
x[0]=0
alpha=0.99
T=1000
shocks=np.random.normal(size=1000,scale=0.007)
for t in range(T):
    x[t+1]=x[t]*alpha + shocks[t]
plt.plot(x, label='AR[1]')
plt.legend()
plt.show()
```



In [212…

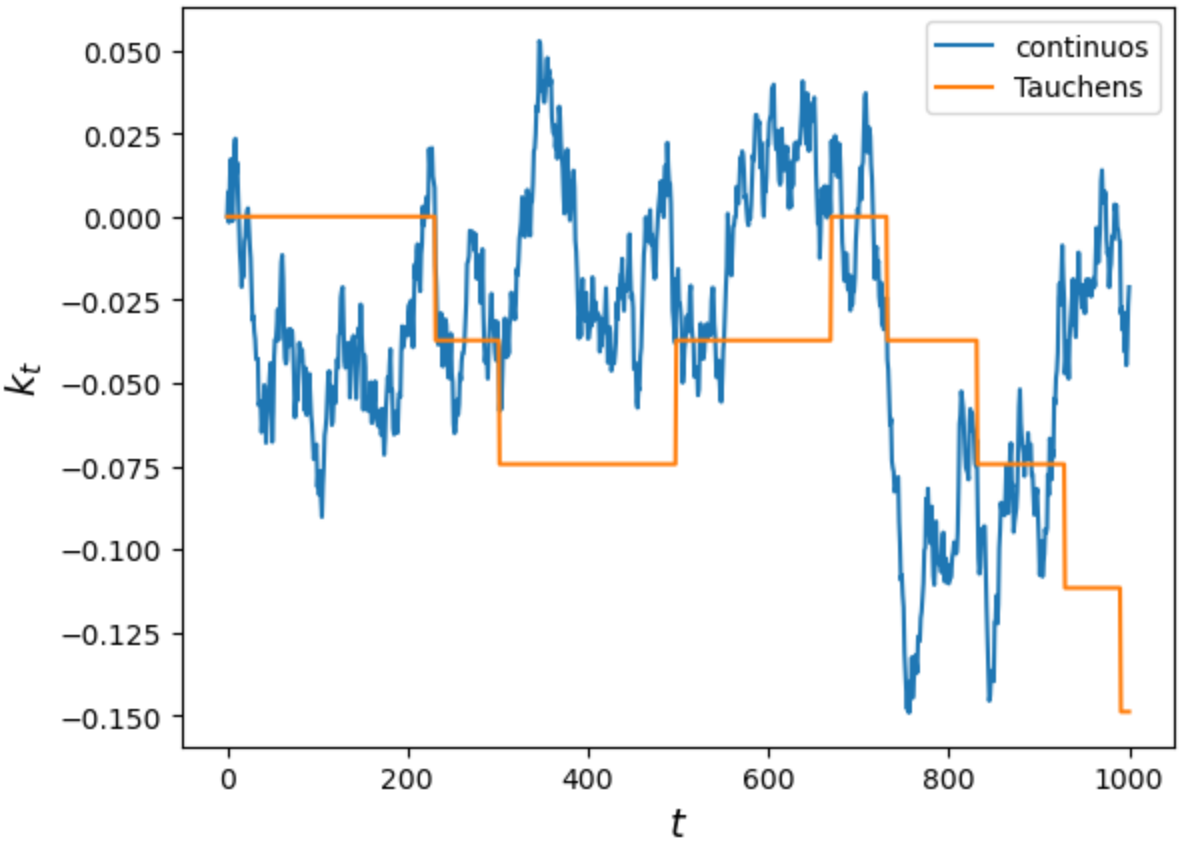
```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trmo[int(current_state[i]), :]), shocks
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])]

plt.plot(simulated_states, label='tauchen')
plt.legend()
plt.show()
```



In [213…

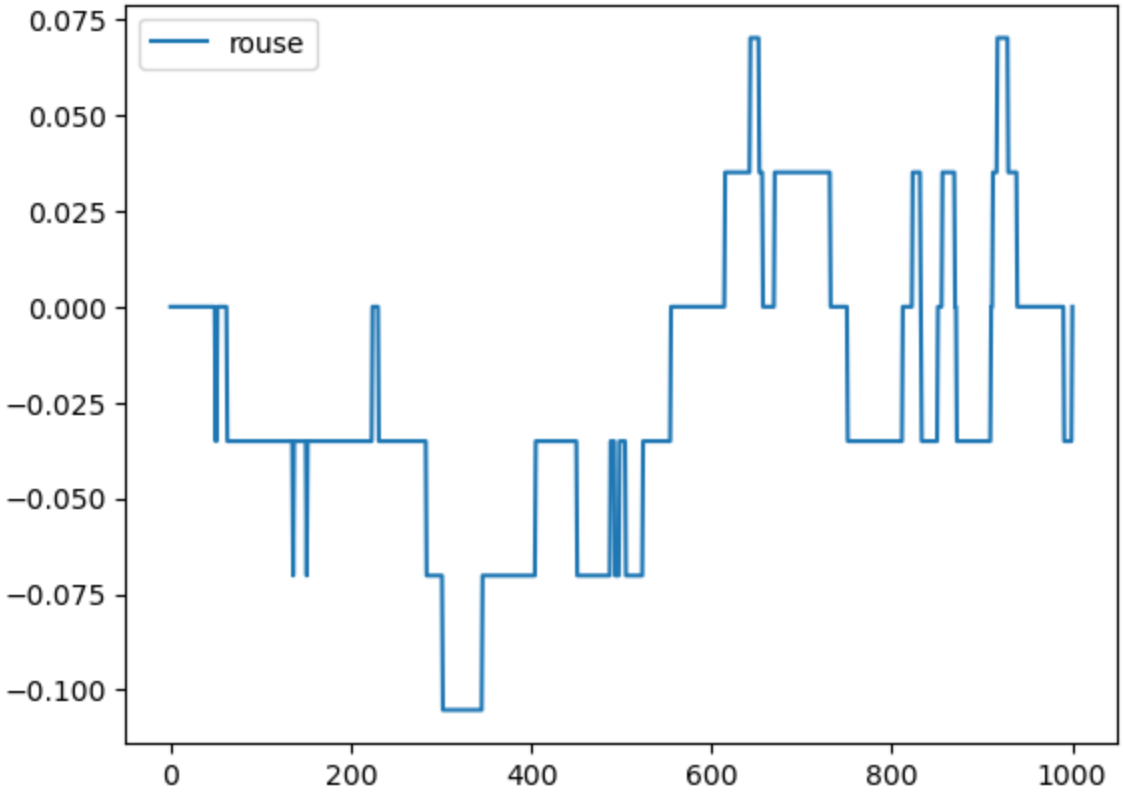
```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states, label='Tauchens')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```



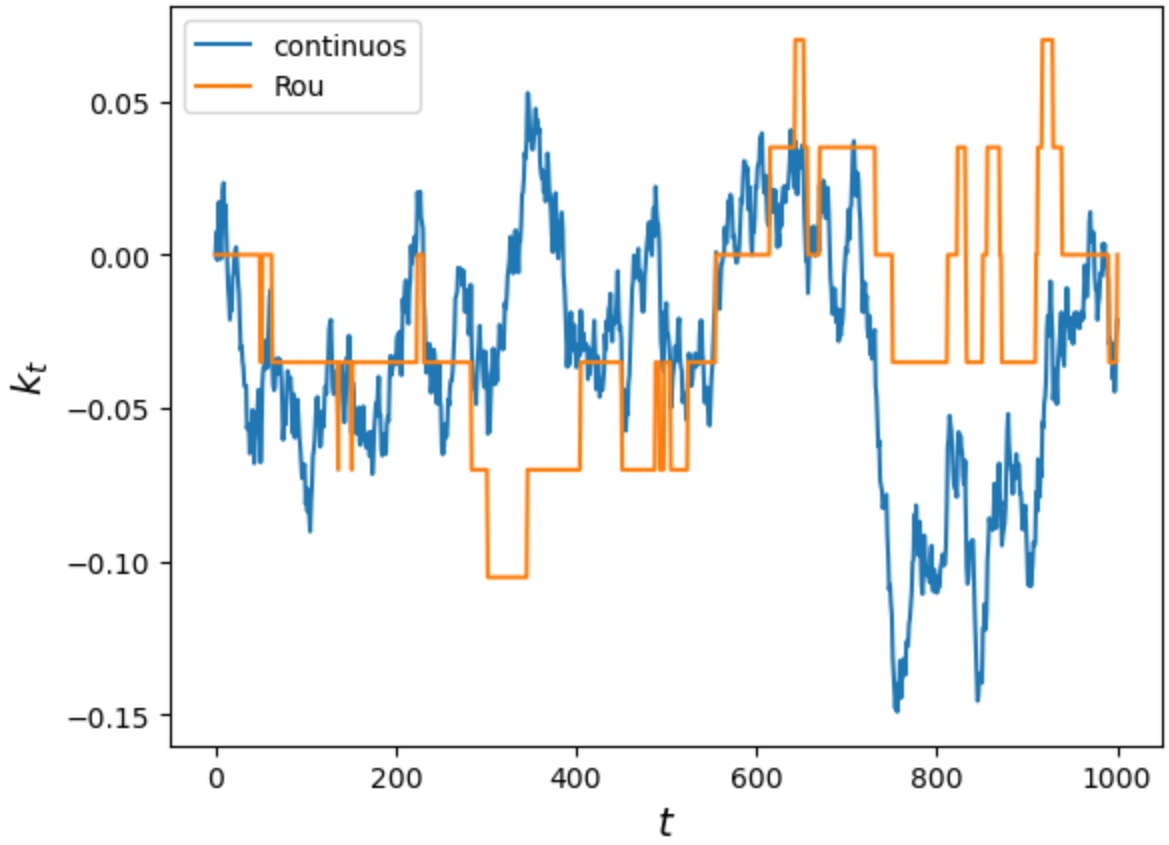
They are really not that close, we will later try again with a bigger n, but now let's see how rou does

In [214…
```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0
for i in range(nsteps-1):
    next_state_index = np.searchsorted(np.cumsum(trm1[int(current_state1[i]), :]), shock
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])]

plt.plot(simulated_states1, label='rouse')
plt.legend()
plt.show()
```



In [215…
```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states1, label='Rou')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```

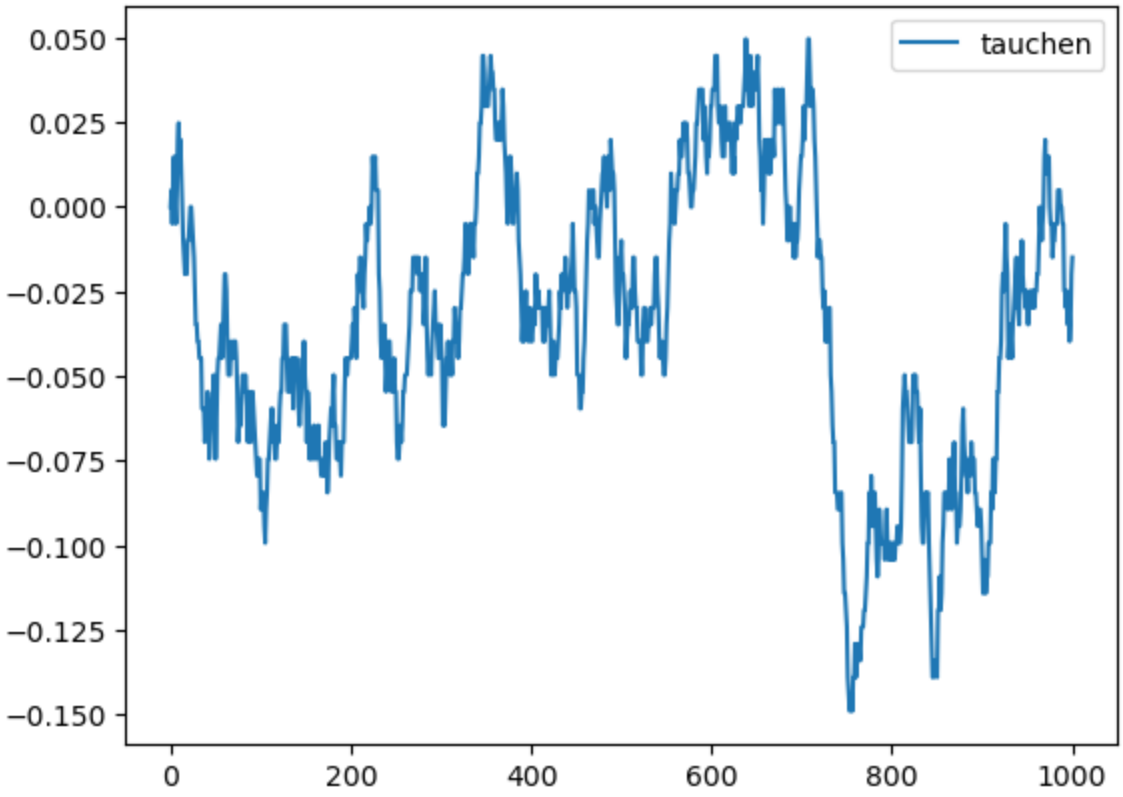


it dosnt do very well either

In [216…
```
rho=0.99
n=61
```

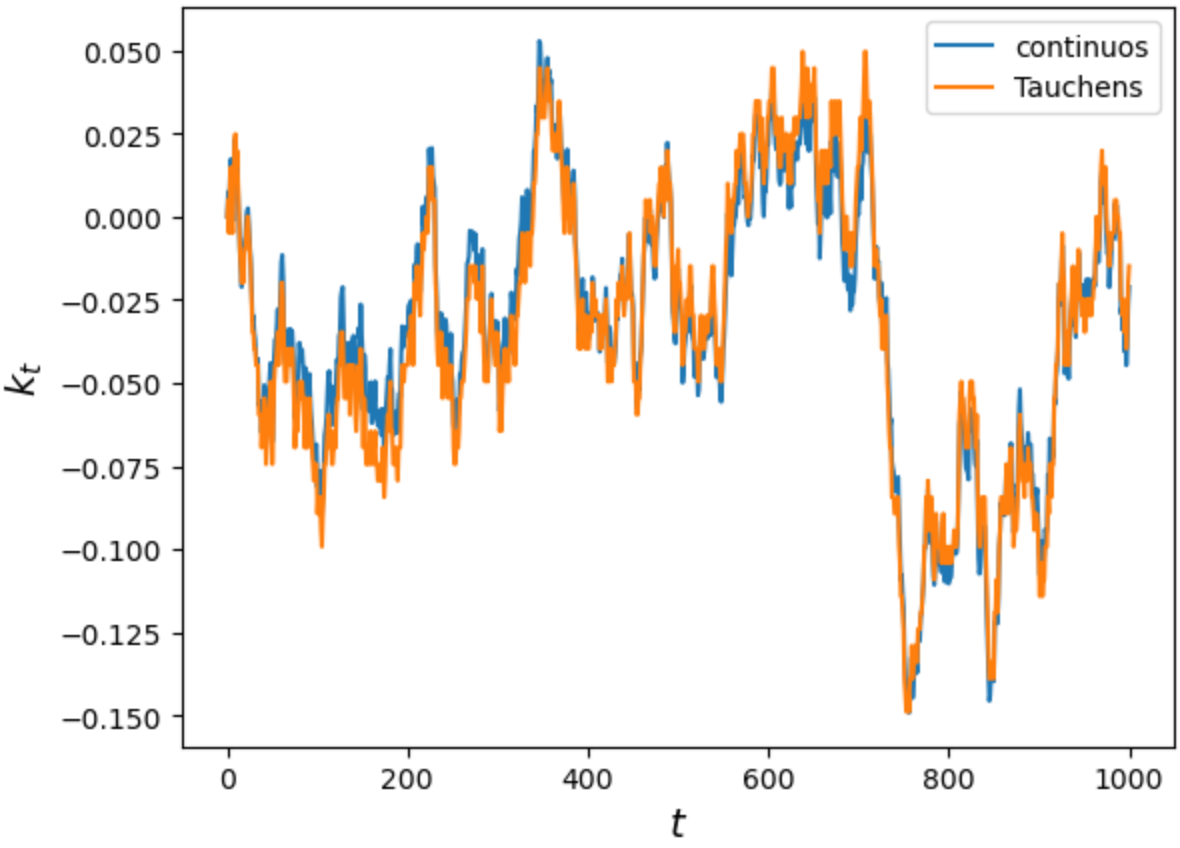
In [217…
```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trmo[int(current_state[i]), :]), shocks
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])]

plt.plot(simulated_states, label='tauchen')
plt.legend()
plt.show()
```



In [218…

```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states, label='Tauchens')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```

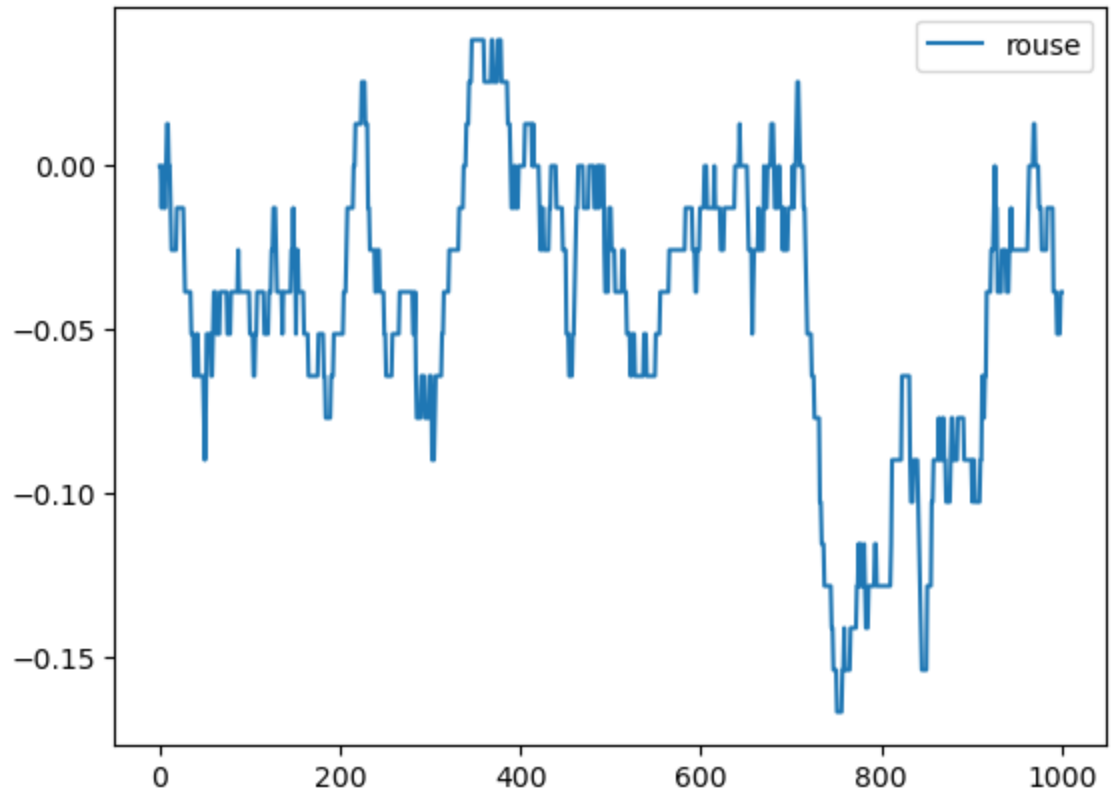


In [219…

```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
```

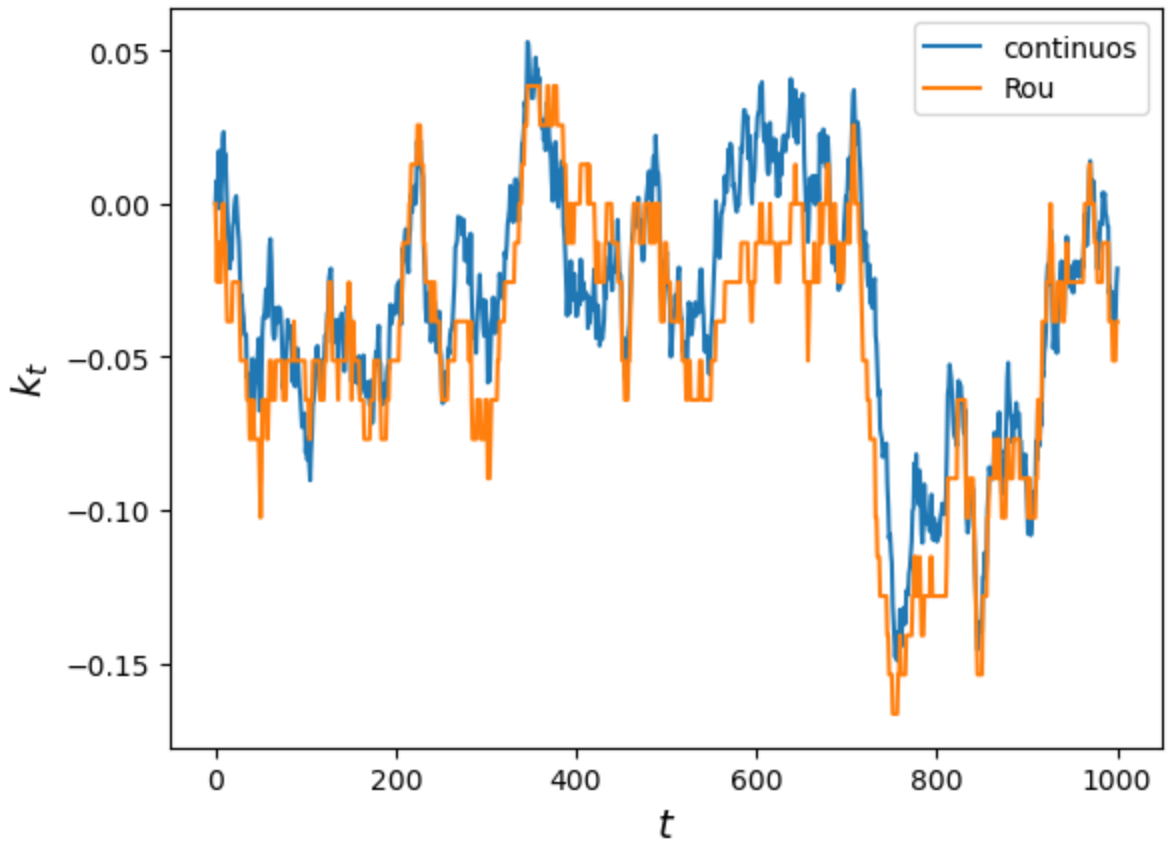
```
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trm1[int(current_state1[i]), :]), shock
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])]

plt.plot(simulated_states1, label='rouse')
plt.legend()
plt.show()
```



In [227…

```
fig, ax = plt.subplots()
ax.plot(x,  label='continuos')
ax.plot(simulated_states1, label='Rou')
ax.set_xlabel('$t$', fontsize=14)
ax.set_ylabel('$k_t$', fontsize=14)
ax.legend()
plt.show()
```



I tried the process with bigger n's and they only started doing good with n>60

In [228…
```
n=9
```

In [229…
```
upp=m*sigma/(1-rho**2)**0.5
low=-upp
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
trm = np.zeros((n, n+1))
trm[:, 1:] = trmo
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trm[int(current_state[i])-1, :]), shock
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])-1]
model = LinearRegression()
model.fit(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshape(-1, 1))
r_sq = model.score(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshap


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.00031312]
Coefficients: [[0.99341806]]
coefficient of determination: 0.9882340581925161
```

In [230…
```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
```

```
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0
trm0 = np.zeros((n, n+1))
trm0[:, 1:] = trm1
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trm0[int(current_state1[i])-1, :]), sho
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])-1]
model = LinearRegression()
model.fit(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].reshape(-1, 1
r_sq = model.score(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].resh


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.00040985]
Coefficients: [[0.98226487]]
coefficient of determination: 0.9644825198495377
```

They both did pretty bad comparing the graphs but the rou method did way better in the regressin, lets try with n=61

In [231…

```
n=61
```

In [232…

```
xgrid = np.linspace(low, upp,n )
trmo=tau(n,rho,sigma,a=low,b=upp)
N = int((n-1)/2)
current_state = np.empty(nsteps+1)
current_state[0] = N
simulated_states = np.zeros(nsteps+1)
simulated_states[0] = 0
trm = np.zeros((n, n+1))
trm[:, 1:] = trmo
for i in range(nsteps):
    next_state_index = np.searchsorted(np.cumsum(trm[int(current_state[i])-1, :]), shock
    current_state[i+1] = next_state_index
    simulated_states[i+1] = xgrid[int(current_state[i+1])-1]
model = LinearRegression()
model.fit(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshape(-1, 1))
r_sq = model.score(simulated_states[1:].reshape(-1, 1), simulated_states[:nsteps].reshap


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.0004849]
Coefficients: [[0.98533937]]
coefficient of determination: 0.9704614940282733
```

In [233…

```
maxi = (sigma**2 / (1 - rho**2))**(1/2) * np.sqrt(n - 1)
mini=-maxi
xgrid1 = np.linspace(mini, maxi, n)
trm1=rouwenhorst(n, rho,sigma)
current_state1 = np.empty(nsteps+1)
current_state1[0] = N
simulated_states1 = np.zeros(nsteps+1)
simulated_states1[0] = 0
trm0 = np.zeros((n, n+1))
trm0[:, 1:] = trm1
for i in range(nsteps):
```

```
    next_state_index = np.searchsorted(np.cumsum(trm0[int(current_state1[i])-1, :]), sho
    current_state1[i+1] = next_state_index
    simulated_states1[i+1] = xgrid1[int(current_state1[i+1])-1]
model = LinearRegression()
model.fit(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].reshape(-1, 1
r_sq = model.score(simulated_states1[1:].reshape(-1, 1), simulated_states1[:nsteps].resh


print('Intercept:', model.intercept_)
print('Coefficients:', model.coef_)
print(f"coefficient of determination: {r_sq}")
```

```
Intercept: [-0.00058317]
Coefficients: [[0.98601416]]
coefficient of determination: 0.971193373866696
```

Tauchens method did even worse in the regression with a bigger n while rou did even better

In [ ]: