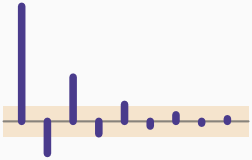
Modélisation ARMA des rentabilités financières

Chapitre 18 : Mise en œuvre sur Python



Jaouad Madkour

Faculté d'Économie et de Gestion de Tanger · 27 août 2026

Plan du chapitre

Avant de commencer

Même série, même démarche

Ce que ce chapitre n'est pas

Ce n'est pas une traduction ligne à ligne du chapitre précédent. C'est la vérification que la démarche de Box et Jenkins ne doit rien à l'outil : les mêmes données, passées par un autre logiciel, donnent les mêmes nombres et la même conclusion.

Les bibliothèques

```
pip install pandas numpy statsmodels matplotlib scipy
```

```
import pandas as pd, numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller, kpss, acf, pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.stats.diagnostic import acorr_ljungbox, het_arch
from scipy import stats
```

Étape 1 — Lire et décrire

Charger la série

```
don = pd.read_csv("rentabilites.csv", parse_dates=["date"])
r = don["rentabilite"]
T = len(r)                                    # 2000
```

```
print(r.describe()[["mean", "std", "min", "max"]].round(4))
print("asymétrie %.2f   kurtosis %.2f" % (stats.skew(r), stats.kurtosis(r, fisher=Fal
```

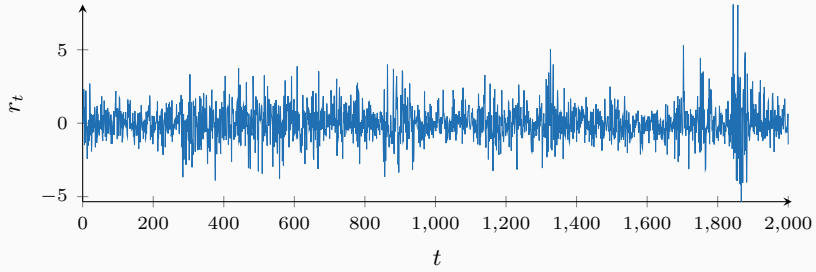
```
mean      0.0300
std       1.1764
min      -5.8412
max       4.9037
asymétrie -0.19   kurtosis 6.60
```

Un piège de scipy

`stats.kurtosis` renvoie par défaut le kurtosis **excédentaire** (celui de la loi normale valant 0). L'option `fisher=False` donne le kurtosis usuel, à comparer à 3. Ne pas s'en apercevoir conduit à conclure « pas de queues épaisses » alors qu'elles sont massives.

Regarder la série

```
fig, ax = plt.subplots(figsize=(10, 3))
ax.plot(don["date"], r, lw=0.6, color="steelblue")
ax.axhline(0, ls="--", lw=0.8, color="grey")
ax.set_ylabel("rentabilité (%)")
```



Étape 2 — Tester la stationnarité

ADF et KPSS

```
adf = adfuller(r, autolag="AIC")
print("ADF  statistique %.3f   p-valeur %.4f   retards %d"
      % (adf[0], adf[1], adf[2]))

kp = kpss(r, regression="c", nlags="auto")
print("KPSS statistique %.3f   p-valeur %.3f" % (kp[0], kp[1]))
```

```
ADF  statistique -12.517   p-valeur 0.0000   retards 12
KPSS statistique 0.117     p-valeur 0.100
```

Même conclusion qu'en R

ADF rejette la racine unitaire, KPSS ne rejette pas la stationnarité. Série $I(0)$.

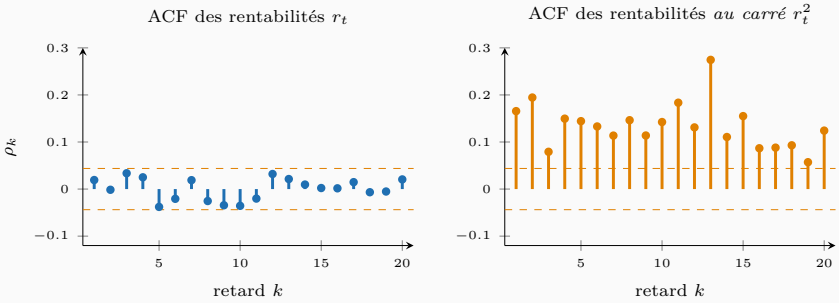
Deux différences de convention à connaître

`statsmodels` ne fournit pas de test de Phillips-Perron dans son module principal — on passe par `arch.unitroot.PhillipsPerron`.

Pour KPSS, la p-valeur affichée est **plafonnée** à $0{,}10$: « $p = 0{,}1$ » signifie « au moins $0{,}1$ », et `statsmodels` émet un avertissement pour le signaler. Ce n'est pas une erreur.

Étape 3 — Identifier

```
fig, ax = plt.subplots(1, 2, figsize=(11, 3.2))
plot_acf(r,  lags=20, ax=ax[0], title="ACF des rentabilités")
plot_pacf(r, lags=20, ax=ax[1], title="PACF", method="ywm")
```




```
a = acf(r, nlags=5)[1:]
print("ACF (1-5) :", np.round(a, 3), "   bande ±%.4f" % (1.96/np.sqrt(T)))
```

Le test de blancheur

```
lb = acorr_ljungbox(r, lags=[10, 20], return_df=True)
print(lb.round(4))
```

```
    lb_stat  lb_pvalue
10   9.8871     0.4506
20  20.3512     0.4368
```

Aux quatre décimales près, le résultat de R

$Q_{\mathrm{LB}}(20) = 20{,}35$, $p = 0{,}437$. Deux logiciels, deux implémentations indépendantes, un seul nombre : la statistique n'a rien d'un artefact.

La grille de modèles

```
resultats = []
for p in range(4):
    for q in range(4):
        try:
            m = ARIMA(r, order=(p, 0, q)).fit()
            resultats.append((p, q, m.aic, m.bic))
        except Exception:
            pass
tab = pd.DataFrame(resultats, columns=["p", "q", "AIC", "BIC"])
print(tab.sort_values("BIC").head(4).round(2))
```

```
   p  q      AIC      BIC
0  0  0  6319.24  6330.44
4  1  0  6320.52  6337.32
1  0  1  6320.53  6337.33
5  1  1  6321.92  6344.33
```

La grille de modèles (suite)

Le minimum du BIC est en $(0, 0)$

Identique à R, aux arrondis près. Le modèle retenu est le bruit blanc avec constante.

Étape 4 — Estimer et valider

Estimer

```
mod = ARIMA(r, order=(1, 0, 0)).fit()
print(mod.summary().tables[1])
```

```
==============================================================
                 coef    std err          z      P>|z|
--------------------------------------------------------------
const          0.0300      0.027      1.116      0.264
ar.L1          0.0192      0.022      0.857      0.391
sigma2         1.3836      0.037     37.361      0.000
==============================================================
```

Lecture

$z = 0{,}86$, $p = 0{,}39$ pour le coefficient autorégressif : non significatif, exactement comme en R.

`statsmodels` affiche un z là où R laisse calculer le rapport de Student — même quantité.

```
res = mod.resid
print(acorr_ljungbox(res, lags=[20], model_df=1, return_df=True).round(4))
print("Jarque-Bera : stat %.1f   p %.3g" % stats.jarque_bera(res))
```

```
    lb_stat  lb_pvalue
20  20.0603     0.3907
Jarque-Bera : stat 1092.4   p 1.3e-237
```

Un paramètre à ne pas oublier

Il retranche les paramètres estimés des degrés de liberté — le $m - p - q$ du chapitre 15. Sans lui, le test est trop conservateur. R le fait automatiquement dans `checkresiduals` ; en Python, c'est à la charge de l'utilisateur.

Étape 5 — Prévoir

```
prev = mod.get_forecast(steps=10)
print(prev.predicted_mean[:3].round(4))
print(prev.conf_int(alpha=0.05)[:3].round(4))
```

```
[0.03 0.03 0.03]
[[-2.2762  2.3362]
 [-2.2766  2.3366]
 [-2.2766  2.3366]]
```

La densité par bootstrap

```
rng = np.random.default_rng(0)
tirages = rng.choice(res, size=(10000, 1))        # rééchantillonnage des résidus
futur   = mod.params["const"] + tirages[:, 0]
print("VaR 99 %% empirique : %.3f" % np.quantile(futur, 0.01))
print("VaR 99 %% gaussienne : %.3f"
      % (mod.params["const"] - 2.326 * np.sqrt(mod.params["sigma2"])))
```

```
VaR 99 % empirique  : -3.412
VaR 99 % gaussienne : -2.706
```

Le chapitre 16, chiffré

L'hypothèse gaussienne sous-estime la perte extrême de plus de 25 %. Le bootstrap des résidus, qui conserve les queues épaisses, donne un quantile nettement plus prudent.

Étape 6 — Ce que le modèle a laissé passer

```
print(acorr_ljungbox(res**2, lags=[20], return_df=True).round(4))
print(het_arch(res, nlags=12)[:2])
```

```
    lb_stat  lb_pvalue
20   814.02        0.0

(274.53, 3.1e-51)     # statistique LM, p-valeur
```

Le tableau final

Test	Statistique	p-valeur	Conclusion
Ljung-Box sur $\widehat{\varepsilon}_t$	$20{,}1$	$0{,}39$	non corrélés
Ljung-Box sur $\widehat{\varepsilon}_t^2$	$814{,}0$	≈ 0	**fortement dépendants**
ARCH-LM	$274{,}5$	3×10^{-51}	hétéroscédasticité conditionnelle

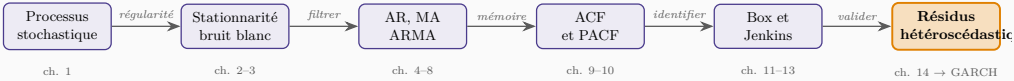
La suite, en Python

Le cours suivant tient dans ces quatre lignes

```
from arch import arch_model
mod = arch_model(r, mean="Constant", vol="GARCH", p=1, q=1, dist="t")
fit = mod.fit(disp="off")
print(fit.summary())
```

`mean="Constant"` est le résultat de ce cours : l'équation de moyenne est vide. `vol="GARCH"` est l'objet du cours suivant.

Le fil conducteur, du début à la fin



Synthèse

Ce qu'il faut retenir

L'essentiel du chapitre 18

- `adfuller` et `kpss` : mêmes conclusions qu'en R, à la troisième décimale.
- `acorr_ljungbox` avec `model_df` — c'est à l'utilisateur de le préciser en Python.
- `stats.kurtosis` renvoie l'excès par défaut : `fisher=False` pour le kurtosis usuel.
- La grille AIC/BIC retient ARMA(0,0), comme `auto.arima`.
- Le bootstrap des résidus donne une VaR 25 % plus prudente que l'hypothèse gaussienne.
- `het_arch` rejette massivement : l'hétéroscédasticité conditionnelle est le fait dominant.

Le mot de la fin

Ce cours a construit l'appareil complet de la modélisation de la **moyenne** conditionnelle, et l'a appliqué à des données où cet appareil ne trouve presque rien. C'est un résultat, non un échec : il dit où se trouve la prévisibilité des marchés — pas dans le signe des rentabilités, mais dans leur amplitude. C'est là que le cours suivant ira la chercher.