MS&E 244: Homework 4

Your Name(s) Here

Introduction

Welcome to MS&E 244: Statistical Arbitrage - Homework 4.

This assignment is due by February 18th, 2026 at 11:59pm Pacific Time and should be done in groups. You may use the same group as your course project. We recommend collaborating via a private GitHub repo.

Instructions for this assignment are below. **Please read all instructions carefully.**

1. Replace "Your Name(s) Here" above in bold with your name, and, if working in a group, your group members' names.
2. Read and run the instructions code in the Setup section. Make sure you understand what it does.
3. Answer the questions in the sections below. Make sure to limit code output to a reasonable length so that the resulting PDF you'll make from this notebook is readable. If the resulting PDF is not readable, keep your lines of code less than 80 characters.
4. Once finished, export or convert the notebook to PDF, using any method you like. For example, you can use the File -> Save and Export Notebook As -> PDF option in Jupyter Lab, you can use the nbconvert command line tool, etc. Whatever displays best is fine. Make sure no code is cut off in the PDF and that sheets are 8.5x11 inch dimensions or similar.
5. Submit the resulting PDF to the Gradescope assignment. If working in a group, only one person needs to submit and select all group members on Gradescope.
6. Zip and submit all files (your .ipynb file and any .py files we provided or you created) to the Canvas assignment. If working in a group, only one member of the group needs to submit to Canvas. Unlike Gradescope, no group members need to be selected on Canvas.

Setup

System Requirements

First, ensure you have the proper system requirements:

1. We will only support Mac or Linux in this course. If you're on Windows, learn to use WSL 2.0 to run Linux on Windows, and then run these commands via WSL.
2. Ensure you have Python 3.10 and pip installed. You also need something to run this Jupyter notebook, whether that's Jupyter Notebook, Jupyter Lab, an IDE like VS Code, or Google

Colab. We recommend collaborating via GitHub and using Jupyter Lab or VS Code to complete this notebook.

Jupyter Lab

To use Jupyter Lab, we'll need to set up the Python virtual environment and install the requirements. We've provided example instructions for doing so below for those using Jupyter Lab. Note that you may need to substitute `pip` with `pip3` depending on your installation. You can learn more by reading the venv docs and the ipykernel docs. See documentation for Jupyter Lab if needed.

```
# Navigate to the root directory containing your course files
cd /path/to/mse244

# Place the requirements.txt file in this directory if it isn't already there
# Put this .ipynb file in this directory or in a subdirectory, e.g. "hw1"

# Create a virtual environment
python3.10 -m venv .venv

# Activate the virtual environment
source .venv/bin/activate

# Install the required packages
pip install -r requirements.txt

# Add the virtual environment to Jupyter
python -m ipykernel install --user --name mse244 --display-name="MSE244"

# Start Jupyter Lab
jupyter lab

# Once in Jupyter, make sure to select the "MSE244" kernel
```

VS Code

If using VS Code, read the docs for Jupyter notebooks in VS Code and virtual environments in VS Code. VS Code can help you create and activate a virtual environment and Jupyter kernel for this notebook through its GUI (though you should also learn how to do these things via the command line). Make sure that you use Python 3.10 to create your virtual environment.

Once setup is complete and you're running within your environment, you can begin executing the cells below.

Imports

[ ]:
```
%load_ext autoreload
%autoreload 2

import warnings
from datetime import datetime
```

```
from typing import Any, Dict, Optional, Tuple, Union

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as mpl_ticker
import seaborn as sns
from scipy import stats
from statsmodels.graphics.tsaplots import plot_acf

# We add any local imports (Python files we've written) here
from hw4 import (
    load_data,
    select_asset_universe,
    estimate_oos_residuals,
    estimate_ou_parameters,
    forecast_residual_returns_ou_signal,
    forecast_returns_noisy_oracle,
    run_portfolio_optimization,
)


# Configure pandas display options
pd.set_option('display.max_columns', 20)
pd.set_option('display.max_rows', 40)
pd.set_option('display.precision', 4)
```

Data

To start with, we'll load the supplied Japanese equity data, preprocess the data, and set up the necessary variables.

Please make sure to look up and learn about any lines you don't understand by reading the Pandas User Guide. This is essential knowledge for anyone working with financial data in Python.

[ ]:
```
config = {
    'NIKKEI_CSV_PATH': 'N225.csv',
    'END_YEAR': 2022
}

prices, returns, tickers, metadata = load_data(config, verbose=True)
```

Outline

In this homework, we'll cover two topics.

First, we'll explore some of the details of the trading strategy proposed in the paper "Statistical Arbitrage in the US Equity Market" by Avellaneda and Lee. The strategy utilizes the Ornstein-

Uhlenbeck (OU) process, which is a mean-reverting stochastic process. The authors propose using this process to model the residual returns of a portfolio of stocks, and then using these models to form trading signals.

Second, we'll explore some details of portfolio optimization, and explore how it could be used as an alternative to the strategy proposed in the paper. We'll suppose that we have a forecasting model for the residual returns, and use this model to optimize a portfolio of the residuals. We'll determine how changing the forecasting model's performance changes the performance of the portfolio optimization.

Here's an outline of what you'll do in this homework:

1. Derive the OU process parameter estimators
2. Estimate OU processes and signals from residuals
3. Forecast residual returns using the OU process
4. Derive a noisy oracle forecasting model for the residual returns
5. Optimize a portfolio of residuals using noisy oracle return forecasts and a simple covariance matrix forecast
6. Explore the effect of changing the forecasting model's performance on the portfolio performance

Throughout the homework, feel free to make any modifications to the code or the strategy that you think will improve performance.

Ornstein-Uhlenbeck Process Review

Introduction

Now that we can estimate residuals, it's time to estimate the parameters of a stochastic process which we'll use to model them. This is a useful step because it allows us to utilize a rule for trading the residuals which is based on the properties of the process. In this homework, we'll use the Ornstein-Uhlenbeck (OU) process to model the residuals. The OU process is a mean-reverting stochastic process which is often used in finance to model interest rates, exchange rates, and other quantities that are expected to revert to a long-term mean.

Although empirical studies have shown that the OU process is a good model for many financial time series, it is important to note that the OU process is not a perfect model. In particular, the OU process makes the following assumptions:

1. The residuals are normally distributed.
2. The process parameters (mean reversion speed, mean, and variance) are constant over time.
3. There are no jumps in the residuals.

All of these assumptions are known to be false in practice. Moreover, the parameter estimates for the OU process can be imprecise and biased in finite samples. However, despite these limitations, the OU process is still a useful tool for understanding mean-reverting time series and can sometimes provide a reasonably good set of insights into their behavior.

Background

To start with, we'll review the definition of the Wiener process W_t, which is a continuous-time stochastic process that is often used in the modeling of the behavior of financial time series. The

Wiener process is defined by the following properties:

- The process starts at zero: $W_0 = 0$.
- The process has independent increments: for any $0 \leq t_1 < t_2 < \cdots < t_n$, the increments $W_{t_2} - W_{t_1}, W_{t_3} - W_{t_2}, \ldots, W_{t_n} - W_{t_{n-1}}$ are independent random variables.
- The process has stationary increments: for any $0 \leq t_1 < t_2$, the increment $W_{t_2} - W_{t_1}$ is normally distributed with mean 0 and variance $t_2 - t_1$.
- The process has continuous paths: with probability 1, the function W_t is continuous in t.

Below we'll define the OU process and derive several properties of it. Important results will be boxed and/or tagged for easy reference.

Definition

The OU process is defined by the following stochastic differential equation:

$$dX_{n,t} = -\kappa_n(X_{n,t} - \mu_n)dt + \sigma_n dW_{n,t}.$$

where $dX_{n,t}$ is the difference, $X_{n,t}$ is the price, $\kappa_n > 0$ is the speed of mean reversion, μ_n is the long-term mean, σ_n is the instantaneous volatility, and $W_{n,t}$ is a Wiener process. The OU process given an initial value $X_0 \sim N(\mu, \sigma^2/2\kappa)$ is stationary and mean-reverting, and it has an equilibrium distribution which is Gaussian. Given a specified initial value X_0, the OU process is actually not stationary, but it converges to the stationary distribution as $t \to \infty$.

Although we have written the OU process in a multivariate form, we will only consider the univariate case in this homework. The multivariate case with nontrivial correlation structure is more complex and its study requires a strong background in stochastic processes. So, we'll drop the subscript n and write the OU process as follows from now on:

$$\boxed{dX_t = -\kappa(X_t - \mu)dt + \sigma dW_t.}$$

Despite the subscripts vanishing, it's understood that each of the parameters are unique to the process being modeled.

Solution

The solution of the OU process can be exactly calculated on the interval $[t_0, t_0 + \Delta t]$ as follows, where X_{t_0} is the initial value at time t_0.

$$X_{t_0+\Delta t} = e^{-\kappa \Delta t} X_{t_0} + (1 - e^{-\kappa \Delta t})\mu + \sigma \int_{t_0}^{t_0+\Delta t} e^{-\kappa(t_0+\Delta t - s)} \, dW(s), \tag{S}$$

Proof. The proof of this formula is a standard exercise in stochastic calculus, and it can be found in many textbooks on the subject. However, we'll go through it here to provide some motivation for what follows. Note that this requires some basic stochastic calculus knowledge—if you don't have this, feel free to skip this section. As a summary, first, we'll rewrite the SDE in a more convenient form, then we'll apply an integrating factor to eliminate the drift term, and finally, we'll integrate the resulting equation to reach the solution formula above.

Step 1: Rewrite the SDE in a more convenient form.

First, we'll rewrite the equation by moving the mean μ onto the left-hand side:

$$dX_t - \kappa(\mu - X_t)\,dt = \sigma\,dW_t \quad \Longrightarrow \quad dX_t + \kappa(X_t - \mu)\,dt = \sigma\,dW_t.$$

Next we will set the long-run mean μ to zero. This is commonly done when solving SDEs to simplify the problem. We can do this by defining a new process Y_t as follows:

$$Y_t := X_t - \mu.$$

Note that now $dY_t = dX_t$, so

$$dY_t = -\kappa Y_t\,dt + \sigma\,dW_t \qquad \text{with} \qquad Y_{t_0} = X_{t_0} - \mu. \tag{1}$$

You may recognize Equation (1) as a linear SDE with constant coefficients.

Step 2: Use the integrating factor method and Itô's lemma to get rid of the drift term.

We'll use the integrating factor method, which is a standard technique for solving linear ordinary differential equations. The idea is to multiply both sides of the equation by a factor which, after the application of Itô's lemma, will eliminate the drift term. Here, the drift term is $-\kappa Y_t\,dt$, so we want to multiply by the integrating factor $e^{\kappa t}$. This will allow us to rewrite the equation in a form that can be integrated directly.

First we'll define the new process with the integrating factor $e^{\kappa t}$ as

$$Z_t := e^{\kappa t} Y_t.$$

Itô's lemma (which can be thought of as the stochastic version of the product rule, if you aren't familiar with it) then gives:

$$dZ_t = \kappa e^{\kappa t} Y_t\,dt + e^{\kappa t} dY_t = \kappa e^{\kappa t} Y_t\,dt + e^{\kappa t}(-\kappa Y_t\,dt + \sigma\,dW_t) = \sigma e^{\kappa t}\,dW_t. \tag{2}$$

Notice the drift terms cancel—this is exactly why we chose this integrating factor. Importantly, Equation (2) now has no drift, so we can integrate it directly.

Step 3: Integrate from t_0 to $t_0 + \Delta t$ to produce the solution equation.

Integrating both sides of (2) we get:

$$\int_{t_0}^{t_0+\Delta t} dZ_t = Z_{t_0+\Delta t} - Z_{t_0} = \sigma \int_{t_0}^{t_0+\Delta t} e^{\kappa s}\,dW_s.$$

Substituting back $Z = e^{\kappa t} Y$ and $Y = X - \mu$ gives:

$$e^{\kappa(t_0+\Delta t)}(X_{t_0+\Delta t} - \mu) - e^{\kappa t_0}(X_{t_0} - \mu) = \sigma \int_{t_0}^{t_0+\Delta t} e^{\kappa s}\,dW_s.$$

Further multiplying by $e^{-\kappa(t_0+\Delta t)}$ we get:

$$X_{t_0+\Delta t} - \mu = e^{-\kappa \Delta t}(X_{t_0} - \mu) + \sigma \int_{t_0}^{t_0+\Delta t} e^{-\kappa(t_0+\Delta t-s)}\, dW_s.$$

Finally, re-arranging to isolate $X_{t_0+\Delta t}$ yields:

$$\boxed{X_{t_0+\Delta t} = e^{-\kappa \Delta t} X_{t_0} + (1 - e^{-\kappa \Delta t})\mu + \sigma \int_{t_0}^{t_0+\Delta t} e^{-\kappa(t_0+\Delta t-s)}\, dW_s}.$$

This is equation (S), as desired. ■

Further Details

Let's first recap each of the terms in the solution equation (S):

- $X_{t_0+\Delta t}$: the value of the process at the future time $t_0 + \Delta t$.
- $e^{-\kappa \Delta t} X_{t_0}$: the part of today's value that survives after mean-reversion during Δt.
- $(1 - e^{-\kappa \Delta t})\mu$: the pull toward the long-run mean μ.
- The Itô integral: the cumulative shock of the Brownian motion over $[t_0, t_0+\Delta t]$, exponentially discounted the further back in time the shock occurred.

A few useful quantities can be calculated from the solution (S).

Mean of the process at time $t_0 + \Delta t$. The expectation of the value of the process at the future time $t_0 + \Delta t$ is:

$$\boxed{\mathbb{E}[X_{t_0+\Delta t}] = e^{-\kappa \Delta t} X_{t_0} + (1 - e^{-\kappa \Delta t})\mu}.$$

Note that the expectation of the Itô integral is zero. If you haven't encountered stochastic integrals, or the Itô integral in particular, before, you can think of them as being like the Riemann integral, where the Riemann sums have stochastic increments. In this case, the Itô integral is defined as the limit of a sum of stochastic increments, where the increments are taken at random times. Here those increments have expectation zero, so the expected value of the Itô integral is zero. This means that the expected value of the process at the future time $t_0 + \Delta t$ is simply the deterministic part of the equation, which is the sum of the mean-reversion term and the long-run mean term.

Variance of the process at time $t_0 + \Delta t$. The variance of this value at this future time can be computed as follows. First, we'll need standard result from stochastic calculus. This result is known as the Itô isometry. It states that that the variance of the Itô integral is equal to the integral of the square of the integrand. The formal statement of the Itô isometry is

$$\text{Var}\left[\int_{t_0}^{t_0+\Delta t} f(s)\, dW(s)\right] = \int_{t_0}^{t_0+\Delta t} f^2(s)\, ds.$$

for any deterministic function $f(s)$ which is square-integrable.

We can apply this result to compute the variance of the Itô integral. In our case, we have $f(s) = e^{-\kappa(t_0+\Delta t-s)}$, which can be integrated by parts. The variance of the Itô integral is then given by (notice the square of the integrand):

$$\int_{t_0}^{t_0+\Delta t} e^{-2\kappa(t_0+\Delta t-s)}\,ds = \frac{1}{2\kappa}(1-e^{-2\kappa\Delta t}).$$

The first two terms on the right-hand side of (S) are deterministic given the information set at time t_0 and thus have zero variance. Thus the variance of the process at the future time $t_0+\Delta t$ is given by:

$$\boxed{\mathrm{Var}[X_{t_0+\Delta t}] = \frac{\sigma^2}{2\kappa}(1-e^{-2\kappa\Delta t})}.$$

Equilibrium distribution of the process. The equilibrium distribution of the OU process is the stationary distribution of the process. We can compute this via methods similar to the mean and variance above. We know that for any $t \geq 0$, starting at time 0,

$$X_t = e^{-\kappa t}X_0 + (1-e^{-\kappa t})\mu + \sigma\int_0^t e^{-\kappa(t-s)}\,dW_s.$$

To get the mean of the distribution, we take expectations and use the fact that $\mathbb{E}[\int_0^t \cdots\, dW_s] = 0$, then take the limit as $t \to \infty$:

$$\mathbb{E}[X_t] = e^{-\kappa t}X_0 + (1-e^{-\kappa t})\mu \xrightarrow[t\to\infty]{} \mu.$$

To get the variance of the distribution, we use the Itô isometry:

$$\mathrm{Var}[X_t] = \sigma^2\int_0^t e^{-2\kappa(t-s)}\,ds = \sigma^2\int_0^t e^{-2\kappa u}\,du = \frac{\sigma^2}{2\kappa}(1-e^{-2\kappa t}) \xrightarrow[t\to\infty]{} \frac{\sigma^2}{2\kappa}.$$

Since X_t is a linear combination of the Gaussian X_0 and the Itô integral (itself Gaussian), X_t is Gaussian for every t. In the limit $t \to \infty$ it therefore converges in law to the Gaussian distribution with mean μ and variance $\frac{\sigma^2}{2\kappa}$:

$$\boxed{X_\infty \sim N\left(\mu, \frac{\sigma^2}{2\kappa}\right)}.$$

We'll denote this equilibrium variance as $\sigma^2_{\text{eq}} := \frac{\sigma^2}{2\kappa}$.

Estimation

When the OU process is observed at discrete intervals (which is the case when we have aggregated data, such as daily or minutely returns calculated from the closing price of each period), the parameters of the process can be estimated using a regression approach. We won't get into how to derive this result in this class, but we'll give the details here. The first fact is that observing the OU process at Δt intervals (where typically e.g. Δt is equal to one day or one minute, etc.) yields an AR(1) process, which can be seen from the solution equation above. The second fact is that

estimation for an AR(1) process can be performed by regressing consecutive lags of the process on one another. In short, for an AR(1) process X_t, the following equation describes the necessary parameters:

$$\boxed{X_{t+1} = a + bX_t + \zeta_{t+1}}$$

where a and b are scalars and ζ_{t+1} is a noise process with mean zero and variance σ_ζ^2. Estimation of the parameters in this equation can be performed in any reasonable way, but as mentioned above, a simple lag-1 regression suffices. After performing this regression, some stochastic process estimation theory (which we won't expound upon) tells us that there exist the following equations for the OU process parameters we'll care about:

$$\boxed{\hat{\kappa} = -\frac{1}{\Delta t}\log(\hat{b}) \qquad \hat{\mu} = \frac{\hat{a}}{1-\hat{b}} \qquad \hat{\sigma}^2 = \frac{\hat{\sigma}_\zeta^2 \cdot 2\kappa}{1-\hat{b}^2} \qquad \hat{\sigma}_{\text{eq}}^2 = \frac{\hat{\sigma}_\zeta^2}{1-\hat{b}^2}}$$

If you're familiar with the stationary variance of an AR(1) process, you'll note the similarity to σ_{eq}^2. Note that these are different from the variance of the next term in the discretized OU process, $X_{t_0+\Delta t}$, which is given by $\hat{\sigma}_\zeta^2$. Also note that these are both different from the parameter σ for the instantaneous volatility in the OU process definition.

Note that we will drop the hats from the parameters in the rest of this document.

Simulation

The OU process can be simulated using the exact formulation at discrete intervals which we covered above. However, for background, we'll cover a more widely applicable method: the Euler-Maruyama method. The Euler-Maruyama method is a numerical method for simulating stochastic differential equations, and is a special case of the more general method of stochastic integration. The core idea is to discretize the stochastic differential equation and then use the discretized equation to simulate the process. We typically do this by breaking the time interval into small steps of size Δt and then iterating the discretized equation over these time steps. We'll use a uniform grid of time steps, which means that the time step size Δt is constant over the entire interval. This is a common approach in numerical simulations of stochastic processes, and it allows us to easily control the accuracy of the simulation by adjusting the size of the time step.

We form the Euler-Maruyama discretization of the OU process by replacing the differentials with finite differences. The discretized equation is thus given by:

$$X_{t+\Delta t} = X_t - \kappa(X_t - \mu)\Delta t + \sigma\sqrt{\Delta t}Z_{t+\Delta t},$$

where we have approximated as follows:

- $dX_t \approx X_{t+\Delta t} - X_t$,
- $dW_t \approx W_{t+\Delta t} - W_t \sim \sqrt{\Delta t}Z_{t+\Delta t}$, where $Z_{t+\Delta t}$ is a standard normal random variable,
- $dt \approx \Delta t$.

The $\sqrt{\Delta t}$ term appears because the variance of the Wiener process is proportional to the time increment. Concretely, by the definition of the Wiener process, $W_{t+\Delta t} - W_t \sim N(0, t + \Delta t - t) \sim$

$\sqrt{\Delta t}N(0,1)$. So, scaling the random variable by the square root of the time increment ensures that the variance of the discretized process matches that of the continuous process.

Note that we won't use this method for estimation (even though it also yields an AR(1) process), as the exact solution of the OU process is computable in closed form and provides more accurate parameter estimates. In fact, performing a simulation using the exact discretization we covered above would yield a more accurate simulation of the OU process than the Euler-Maruyama discretization. The Euler-Maruyama method provides a less fidelitous numerical approximation, and it introduces additional error into the estimates. This error can be significant, especially for longer time intervals. Instead, the Euler-Maruyama method is often used for simulating stochastic processes when an analytical solution is not available or when the process is too complex to be solved analytically. When you have a closed-form solution, as we do here, it's generally better to use that solution for estimation and simulation. However, it's still useful to know how to implement the Euler-Maruyama method, as it can be applied to a wider range of stochastic processes.

Use with Residuals

There are a few things we need to keep in mind to use OU processes to model residuals.

Price of a residual. We will define the "price" of X_t as the cumulative sum of residual returns. This is of course different from the normalized price we studied in Gatev et al., and the usual definition of a price (which involves the cumulative product), but it is more convenient for our purposes given the definition of the OU process model.

Time-varying parameters. In practice, we often allow the parameters of the OU process to be time-varying. The rationale for this choice arises from the fact that the parameters of the residual process usually change over time. For example, the speed of mean reversion might change if the market environment changes. The parameters of the OU process can be estimated using the estimation procedure above at each time point over a rolling lookback window. The lookback period L is a hyperparameter to be tuned. In this case, we simply replace all of the quantities in the Estimation section with rolling lookback windows and calculate the parameters at each time point t.

Filtering residuals. Real factor model residuals may not be (and often are not) mean-reverting. This poses an obvious problem for the OU process model. To ameliorate the problem, we can filter out the residuals which are not mean-reverting. This can be done by estimating the parameters of the OU process and ensuring that the estimated parameters obey the condition $0 \leq \widehat{b} < 1$ (or equivalently, $0 < \widehat{\kappa} < \infty$). If the estimated parameters for a specific residual do not imply mean reversion, then we won't trade that specific residual.

References

1. Avellaneda, M., & Lee, J. H. (2010). Statistical arbitrage in the US equities market. Quantitative Finance, 10(7), 761-782.
2. Oksendal, B. (2013). Stochastic differential equations: an introduction with applications. Springer Science & Business Media.
3. Uhlenbeck, G. E., & Ornstein, L. S. (1930). On the theory of the Brownian motion. Physical Review, 36(5), 823.
4. Vasicek, O. (1977). An equilibrium characterization of the term structure. Journal of Financial Economics, 5(2), 177-188.

5. Ricciardi, L. M., & Sato, S. (1988). First-passage-time density and moments of the Ornstein-Uhlenbeck process. Journal of Applied Probability, 25(1), 43-57.

Examples

This example section is optional and ungraded.

Below, we write a function called `simulate_ou_process` which simulates the Ornstein-Uhlenbeck process. The function takes the following parameters:

- `kappa`: The speed of mean reversion.
- `mu`: The long-term mean.
- `sigma`: The volatility.
- `dt`: The time step for the simulation.
- `T`: The number of time steps to simulate.
- `X0`: The initial value of the process.
- `seed`: An optional seed for the random number generator.

Using this function, we can simulate some OU processes to give you some intuition about these parameters.

We invite you to explore the following default parameters and plot the resulting process(es):

Default parameters unless otherwise specified: $\kappa = 0.05, \mu = 0.0$, $\sigma = 0.5$, $X_0 = 0.0$, $T = 252$, $\Delta t = 1$.

Scenarios:

- $\kappa \in \{-0.01, 0.0, 0.01, 0.05, 0.1\}$. Plot all five on the same plot with different colors and a legend. Make two plots for two different random seeds.
- $\mu \in \{-10, 0.0, 10\}$. Plot all three on the same plot with different colors and a legend. Make two plots for two different random seeds.
- $\sigma \in \{0.1, 0.5, 1.0\}$. Plot all three on the same plot with different colors and a legend. Make two plots for two different random seeds.
- $\Delta t \in \{0.001, 0.01, 0.1, 1.0\}$. Plot all four on the same plot with different colors and a legend. Make two plots for two different random seeds.

Note that the function below allows for time-varying parameters which you can also supply and explore.

[ ]:
```python
def simulate_ou_process(kappa, mu, sigma, dt, T, X0, seed):
    """
    Simulate an OU process given parameters.
    """
    np.random.seed(seed)

    def dW(dt: float) -> float:
        return np.random.normal(loc=0.0, scale=np.sqrt(dt))

    # implement the Euler-Maruyama scheme
    N = T / dt
    times = np.arange(0, T + dt, dt)
```

```
    assert times.size == N + 1, "T must be a multiple of dt"
    x = np.zeros(times.size)
    x[0] = X0
    for i in range(1, times.size):
        t = (i - 1) * dt
        xold = x[i - 1]
        x[i] = xold - kappa(xold, t) * (xold - mu(xold, t)) * dt + sigma(xold,␣
 ↪t) * dW(dt)

    return times, x

def plot_simulations(num_sims: int):
    """
    Plot `num_sims` simulations in one plot.
    """
    plt.figure(figsize=(12,6))
    for i in range(num_sims):
        plt.plot(*simulate_ou_process(
            lambda x,t: KAPPA,
            lambda x,t: MU,
            lambda x,t: SIGMA,
            DT,
            T,
            X0,
            SEED+i if isinstance(SEED, int) else None
        ))
    plt.title(f"Simulated OU Process with {num_sims} Simulations")
    plt.grid(True)
    plt.xlabel("Time t")
    plt.ylabel("X(t)")
    plt.show()


NUM_SIMS = 4
KAPPA = 0.05
MU = 0
SIGMA = 0.5
DT = 1
T = 252
X0 = 0
SEED = None  # change this to an integer to get reproducible results

plot_simulations(NUM_SIMS)
```

Questions

1 Question 1

(3 points)

Derive the estimators for the OU process parameters κ, μ, and σ_{eq} in terms of the estimated parameters of the AR(1) process given in the Estimation section above.

Hint: Use the solution equation and put it into the form of a discretized AR(1) process. Use the AR(1) process's estimated parameters to match the values for the solution equation to derive the estimator equations. Equations for κ and μ are easy to derive, but σ_{eq} is just a touch more involved. You can use the equation for the equilibrium distribution above for help. Try not to confuse the different σ's in the equations.

Solution:

---

2 Question 2

(5 points)

Now you'll implement these estimators in Python. The function `estimate_ou_parameters` in `hw4.py` takes a time series of residuals and estimates the parameters of the OU process using the method described above.

As a rough guide, here is what the function should do:

1. Cumulatively sum the residual returns to get the residual, and create a 1-lagged residual.
2. Compute the mean and variance of the residuals.
3. Compute the mean and variance of the lagged residuals.
4. Compute the covariance between the residuals and the lagged residuals.
5. Compute the slope and intercept of the regression line.
6. Compute the parameters of the OU process using the equations from the Estimation section.
7. Compute the R^2 value of the regression.
8. Create an integer-valued mask (1 for True and 0 for False) for the residuals that are not mean-reverting where: `mask = (betas > 0.0) & (betas <= config['B_THRESHOLD']) & (r_squared >= config['R_SQUARED_THRESHOLD'])`
9. Create the Avellaneda-Lee "s-score" signal, defined as signal $= (X_t - \mu)/\sigma_{\text{eq}}$.
10. Set the parameters `mu`, `sigma`, `kappa`, and `signal` to zero for the residuals that have a mask value of 0.
11. Return everything you've computed in a DataFrame, where each row corresponds to a residual.

Please complete the `estimate_ou_parameters` function in `hw4.py` and paste the function below. In your implementation, use vectorized operations and avoid using loops. Our implementation of the code to complete is about 30 lines, but yours may be longer or shorter depending on your style.

[ ]:
```
# Your function here
```

Now, to check that it's working, we'll estimate residuals and plot the process parameters and signals for a given residual.

Note that here we select the asset universe using the *entire* set of future prices/returns, which induces lookahead bias. This is to simplify this homework and allow us to write different parts of this analysis in different questions, but in practice you should never use future data to select the asset universe. Instead, you should use the past data to select the asset universe and then use the same past data to estimate parameters, compute signals, forecast returns, optimize the portfolio, make the trades, etc.

It's also worth noting that in some cases the universe can be scoped down using a less strict filter which looks at future data, and then a stricter point-in-time filter can be applied at each universe selection time point (e.g. each month); however, one has to be very careful to ensure that the stricter filter is contained within the less strict filter and does not use future data itself. In any case, this is not what's done below.

[ ]:
```
# Select the universe of assets to be used in the factor models
config = {
    'LOOKBACK_PERIOD': len(returns) - 1,
    'FILTER_MAX_ABS_RETURN': 0.5
}
valid_prices, valid_returns, valid_stocks = select_asset_universe(
    prices,
    returns,
    prices.index[-1],
    config
)

config.update({
    'USE_INTERCEPT': False,
    'FACTOR_MODEL': 'pca',  # 'pca' or 'sector'
    'N_FACTORS': 15,  # Number of factors for PCA
    'RESIDUAL_ESTIMATION_LOOKBACK_DAYS': 60,
    'FACTOR_ESTIMATION_FREQUENCY_DAYS': 5,
    'VERBOSE': False,
})

T = len(valid_returns)
N = len(valid_returns.columns)

(
    residual_returns,
    residual_prices,
    comp_mtxs,
    estimation_dates,
    alphas,
) = estimate_oos_residuals(
    valid_returns,
```

```
    valid_prices,
    metadata,
    config
)
```

[ ]:
```
config.update({
    'OU_ESTIMATION_LOOKBACK_DAYS': 60,
    'B_THRESHOLD': np.exp(-1/30),  # from the paper
    'R_SQUARED_THRESHOLD': 0.25,
})

signals = []

for t in range(config['RESIDUAL_ESTIMATION_LOOKBACK_DAYS'], T):
    resids_lookback = residual_returns.
 ↪iloc[t-config['OU_ESTIMATION_LOOKBACK_DAYS']:t]
    estimation_date = resids_lookback.index[-1]
    if len(resids_lookback) < config['OU_ESTIMATION_LOOKBACK_DAYS']:
        continue
    if config.get('VERBOSE', False) and t %␣
 ↪config['OU_ESTIMATION_LOOKBACK_DAYS'] == 0:
        print(f"Estimating OU parameters at time {t} (date {resids_lookback.
 ↪index[-1]})...")
    # Estimate OU parameters for the residuals
    signals_t = estimate_ou_parameters(resids_lookback, config)
    # Deal with case where intercept is used (though we won't use the intercept␣
 ↪in this homework)
    if config['USE_INTERCEPT']:
        if estimation_date in alphas.index:
            signals_t['alpha'] = alphas.loc[estimation_date]
        else:
            latest_alpha_date = alphas.index[alphas.index < estimation_date].
 ↪max()
            signals_t['alpha'] = (
                alphas.loc[latest_alpha_date]
                if latest_alpha_date is not None
                else np.nan
            )
        signals_t['signal_mod'] = (
            signals_t['signal']
            - signals_t['alpha'] / (signals_t['sigma'] * signals_t['kappa'])
        )
        signals_t.loc[signals_t['mask'] == 0, 'signal_mod'] = 0  # Set␣
 ↪signal_mod to 0 where mask is 0
    signals_t['date'] = estimation_date
    signals_t = signals_t[['date'] + signals_t.columns[:-1].tolist()]
    signals.append(signals_t)
```

```
signals_df = pd.concat(signals)
# Make multiindex from date column and index
signals_df.index.name = 'ticker'
signals_df.set_index(['date', signals_df.index], inplace=True)
```

[ ]:
```
# Choose some asset to plot
ticker_to_plot = valid_stocks[-1]
company_name = metadata.loc[ticker_to_plot, 'Company']

# Plot the factor model predictions vs the actual cumulative returns for the␣
 ↪ticker
with plt.style.context('ggplot'):
    fig, axes = plt.subplots(nrows=5 if config['USE_INTERCEPT'] else 4,
                             ncols=1, figsize=(12, 12), sharex=True)
    axes[0].set_title(f"Out-of-Sample {config['FACTOR_MODEL'].upper()} "
                      f"Factor Model Residual and OU Parameters for␣
 ↪{company_name}")

    axes[0].plot(residual_returns[ticker_to_plot].cumsum(), label='Residual␣
 ↪Process')
    axes[0].set_ylabel('Residual Process')
    axes[0].legend()
    axes[0].yaxis.set_major_formatter(mpl_ticker.PercentFormatter(xmax=1))

    axes[1].plot(signals_df.xs(ticker_to_plot, level='ticker', drop_level=True).
 ↪kappa,
                 label='Kappa', color='orange')
    axes[1].set_ylabel('Kappa')
    axes[1].legend()

    axes[2].plot(signals_df.xs(ticker_to_plot, level='ticker', drop_level=True).
 ↪sigma,
                 label='Sigma', color='green')
    axes[2].set_ylabel('Sigma')
    axes[2].legend()
    axes[2].yaxis.set_major_formatter(mpl_ticker.PercentFormatter(xmax=1))

    if config['USE_INTERCEPT']:
        axes[3].plot(signals_df.xs(ticker_to_plot, level='ticker',␣
 ↪drop_level=True).signal_mod,
                     label='Signal (Modified)', color='purple')
        axes[3].set_ylabel('Signal')
        axes[3].legend()
        axes[4].plot(alphas[ticker_to_plot], label='Intercept (Daily Alpha)',␣
 ↪color='red')
```

```
        axes[4].set_ylabel('Alpha')
        axes[4].legend()
        axes[4].yaxis.set_major_formatter(mpl_ticker.PercentFormatter(xmax=1))
    else:
        axes[3].plot(signals_df.xs(ticker_to_plot, level='ticker',␣
↪drop_level=True).signal,
                     label='Signal', color='purple')
        axes[3].set_ylabel('Signal')
        axes[3].legend()

    axes[-1].set_xlabel('Date')

    plt.tight_layout()
    plt.show()
```

3 Question 3

(8 points)

In the Avellaneda & Lee paper, trading is performed in a "bang-bang" fashion, meaning that the signal is used to determine when to buy and sell the residual. No trading in the residual occurs in between these times. However, to assess the performance of the signal itself (apart from position sizing dynamics), or use it in a portfolio optimization context, it's nice to have a forecast of the residual returns.

To do this, we'll regress the future residual returns on the signal. Our model for the expected future residual returns is given by:

$$\mathbb{E}[r_{t+1:t+H} | \, \text{signal}_t] = \beta_0 + \beta_1 \cdot \text{signal}_t$$

where we define

$$r_{t+1:t+H} := \left[\prod_{\tau=1}^{H} (1 + r_{t+\tau}) \right] - 1$$

as the H-period cumulative return, β_0 is the intercept, and β_1 is the slope of the regression line.

We'll train one regression model across all the residuals, and use a rolling window to compute the regression coefficients. We'll refit the regression coefficients β_0, β_1 periodically.

Complete the function `forecast_residual_returns_ou_signal` in `hw4.py` to implement this regression. The function should take the following parameters:

- `residual_returns`: The DataFrame of valid residual returns.
- `signals_df`: The DataFrame of signals.
- `config`: The configuration dictionary containing the following keys:
  - `RETURN_FORECAST_HORIZON`: The forecast horizon (H) in days.
  - `RETURN_FORECAST_LOOKBACK`: The lookback period in days for the rolling regression.
  - `RETURN_FORECAST_REFIT_PERIOD`: The refit period for the regression coefficients, such as 'M' for monthly (no need to implement periods).

The function should return a DataFrame of the same shape with the same index and columns as `residual_returns` which contains the forecasted residual returns based on the signals. The forecasted returns should be computed using a linear regression of the future residual returns on the signals, using a rolling window of size `RETURN_FORECAST_LOOKBACK`. Fill any missing values in the forecasted returns (e.g. which are inside the first `RETURN_FORECAST_LOOKBACK` days) with NaN.

To be explicit, here's what our implementation does:

1. Extract configuration parameters from the config dictionary, create any empty DataFrames to store forecasted returns, future cumulative returns, and beta coefficients.
2. Extract signals and masks from the multiindex signals DataFrame into regular non-multiindex DataFrames.
3. Precompute future cumulative returns for all dates.
4. Loop through each date in the residual returns DataFrame:
   - Skip dates with insufficient lookback or future horizon data
5. Determine when to refit the regression model. Convert current date to a period (e.g., monthly) using the refit period from config. Only refit when entering a new period (e.g., new month).
6. If refitting, collect training data:
   - Get past signals, masks, and future returns for the lookback period
   - Only include tickers with valid mask (=1), non-NaN signals, and non-NaN future returns
   - Stack all valid (signal, future return) pairs across all tickers and dates into X and y arrays
7. Fit the linear regression model (we use `scipy.stats.linregress`, but you can use whatever you want):
   - Calculate slope, intercept, and r-value
   - Store coefficients in the beta coefficients DataFrame
8. Make predictions for each ticker using the fitted model:
   - For each ticker with a non-NaN signal, calculate forecast = beta_0 + beta_1 * signal
   - Store the forecast in the forecasted returns DataFrame
9. Return three DataFrames:
   - Forecasted returns
   - Future cumulative returns (actual future returns for evaluation)
   - Beta coefficients (regression parameters for each refit date)

Paste your implementation of the `forecast_residual_returns_ou_signal` function below. Our implementation of the code to complete is about 50 lines, but yours may be longer or shorter depending on your style.

[ ]:
```
# Your function here
```

Finally, we'll forecast the returns and perform some analysis on the forecasted returns. There are many ways to do this, but we'll focus on the information coefficient (IC) of the forecasted returns, which is a measure of the quality of the forecast. The IC is defined as the correlation between the forecasted returns and the actual returns. A higher IC indicates a better forecast. The IC can be computed for a single asset (N.B.: a residual is an asset) using the following equation:

$$\text{IC} := \frac{\text{Cov}(\hat{r}_t, r_t)}{\sigma_{\hat{r}_t}\sigma_{r_t}}$$

where r_t is the actual return at time t, $\hat{r}_t$ is the forecasted return at time t, and σ_{r_t} and $\sigma_{\hat{r}_t}$ are

the standard deviations of the actual and forecasted returns, respectively.

The IC can take values between -1 and 1, where 0 means no signal and ± 1 means perfect signal (though in the case of -1, the signal is perfectly wrong; we'll just adopt the convention of referring to the positive end of the spectrum). A range for the IC might be between 0.0 and 0.3, depending on the period, market, horizon, and other factors. Typical values for monthly equity returns in developed markets for firms engaged in quantitative trading might fall between roughly 0.0 and 0.20, but can vary significantly depending on the specific market, time period, and skill of the firm.

Note that although we give the IC for the next time step, it can be computed for any time step. For example, if we expect to hold over a horizon H, we can compute the IC for the forecasted returns between time $t+1$ and $t+H$ (inclusive) using the same equation.

Below we run the function and calculate the information coefficient of the return forecasts separately for each residual. We set the forecast horizon to about one month (20 trading days), the lookback period to about three months (60 trading days; same as our other estimation periods), and refit the forecast model every month. We then print the median IC and plot a histogram of the ICs. Don't be surprised if the median IC isn't very high or is insignificant.

[ ]:
```
config.update({
    'RETURN_FORECAST_HORIZON': 20,
    'RETURN_FORECAST_LOOKBACK': 60,
    'RETURN_FORECAST_REFIT_PERIOD': 'M',
    'VERBOSE': False,
})

(
    ou_forecasted_residual_returns,
    future_cumul_returns,
    beta_df,
) = forecast_residual_returns_ou_signal(
    residual_returns,
    signals_df,
    config,
)
```

[ ]:
```
# Calculate the information coefficient between the forecasted and actual␣
 ↪returns
ic = ou_forecasted_residual_returns.corrwith(future_cumul_returns)
print("Median IC:", ic.median().round(4))
ic.hist(bins=20)
plt.axvline(0, color='red', linestyle='--', label='Zero IC')
plt.axvline(ic.median(), color='blue', linestyle='--', label='Median IC')
plt.legend()
plt.title('Information Coefficients of Residuals')
plt.show()
```

4 Question 4

(8 points)

Using the OU process to model the residuals is a decent idea, but in practice, residuals are much more complex. It's more ideal to have a forecasting model which can predict the residual returns directly, without parametric assumptions. In this question, we'll derive a noisy oracle forecasting model for the residual returns. The noisy oracle forecasting model will provide forecasts of returns which have a certain level of signal.

"Oracle" here means that we use the true future returns as part of our forecast. So, this is a forecasting model which is as good as you specify it to be. Using this model, we can perform a backtest or optimize a portfolio to see how well it performs under a particular level of forecasting skill. This lets us vary the forecasting skill of the model to see what level of skill is needed to achieve a certain level of portfolio performance. That's nice, because it gives us a number to hit when we're trying to improve the performance of our forecasting model.

The metric with which we will parameterize the signal is the information coefficient (IC) of the forecast. We'll choose the IC to be a parameter of the model, and then use this parameter to derive the level of noise which we add to the true returns to get our forecasted returns.

4.1 Part 1

Now we'll derive the noisy oracle forecasting model for the residual returns. Your task is to find the amount of Gaussian noise to add to the future returns to achieve a desired IC. The model is given by the following equation for the noisy oracle forecast:

$$\hat{r}_{t+\Delta t} = r_{t+\Delta t} + \sigma_\eta Z_{t+\Delta t}$$

where $\hat{r}_{t+\Delta t}$ is the return forecast, $r_{t+\Delta t}$ is the true return, σ_η is the standard deviation of the noise, and $Z_{t+\Delta t}$ is a standard normal random variable. The objective is to derive an expression for the optimal noise level σ_η which achieves a given IC.

Derive the expression for the optimal standard deviation of the noise σ_η^2.

Hint: Set the model to a particular IC and solve for the noise level σ_η.

Solution:

4.2 Part 2

We will furthermore find the optimal scaling factor for this forecast to give it a normalized scale. This will be done by finding the optimal scaling factor α which minimizes the mean squared error (MSE) of the forecast. The MSE is given by the following equation:

$$\text{MSE} := \mathbb{E}\left[\left(\alpha\hat{r}_{t+\Delta t} - r_{t+\Delta t}\right)^2\right].$$

We will solve the problem

$$\min_{\alpha} \{\text{MSE}\} = \min_{\alpha} \mathbb{E}\left[\left(\alpha \hat{r}_{t+\Delta t} - r_{t+\Delta t}\right)^2\right].$$

Derive the optimal scaling factor α which minimizes the MSE.

Hint: it's a pretty simple function of the IC.

Solution:

4.3 Part 3

Complete the function `forecast_returns_noisy_oracle` in `hw4.py` to implement the noisy oracle forecasting model.

To be clear, here is what our function does:

1. Takes residual returns and config parameters
2. Calculates future cumulative returns over a specified horizon. As an approximation, we just use the mean of the returns over the horizon to represent this (consequently, the function runs a lot more quickly).
3. Calculates the variance of the returns, σ_η^2 (the variance of the random noise scaled by the information coefficient), and α (the optimal coefficient).
4. Creates noise with standard deviation σ_η.
5. Returns `alpha * (future_returns + noise)` as the forecast.

Paste your implementation of the function below. In your implementation, use vectorized operations and avoid using loops. Our implementation of the code to complete is less than 10 lines, but yours may be longer or shorter depending on your style.

[ ]:
```
# Your function here
```

We'll now use this function to create the residual return forecasts and compute the information coefficient of the forecasts to make sure that the IC is as expected. The median IC should be near 0.10, though the histogram should have some spread.

[ ]:
```
config.update({
    'RETURN_FORECAST_HORIZON': 21,
    'INFORMATION_COEFFICIENT': 0.10,
    'SEED': 42,
})

oracle_forecasted_residual_returns = forecast_returns_noisy_oracle(
    residual_returns,
    config
)

# calculate the information coefficient again
ic = oracle_forecasted_residual_returns.corrwith(future_cumul_returns)
```

```
ic.hist(bins=20)
print("Median IC:", ic.median().round(4))
plt.title('Information Coefficients of Residuals')
plt.show()
```

5 Question 5

(3 points)

We will now use the residuals and the noisy oracle forecasting model to optimize a portfolio of residuals. The portfolio optimization problem is given by the following equation:

$$\max_{\mathbf{w}} \left\{ \mathbf{w}^T \hat{\mathbf{r}} - \frac{\gamma}{2} \mathbf{w}^T \hat{\Sigma} \mathbf{w} \right\}$$

where $\mathbf{w}$ is the vector of portfolio weights, $\hat{\Sigma}$ is the expected covariance matrix of the residual returns, and $\hat{\mathbf{r}}$ is the vector of expected returns. The parameter γ controls the risk aversion of the portfolio, with higher values of γ leading to more risk-averse portfolios. Here, we use $\gamma = 1$. The objective is to maximize the portfolio's expected return while minimizing the portfolio's γ-weighted expected variance.

The optimization problem is subject to the following constraints:

$$\mathbf{1}^T \mathbf{w} = 0, \quad \|\mathbf{w}\|_\infty \leq 0.05, \quad \|\mathbf{w}\|_1 \leq 1, \quad \|\mathbf{w}_t - \mathbf{w}_{t-1}\|_1 \leq 0.25$$

The first constraint ensures that the portfolio weights in residual-space sum to zero and enforces long-short neutrality, the second constraint ensures that the maximum allocation we make to any one residual is at most 5% in magnitude, the third constraint ensures that the total leverage of the portfolio in residual-space is bounded by 1, and the fourth constraint ensures that the change in weights in residual-space from one time step to the next (which is a proxy for turnover) is bounded by 25%. We've set these bounds a priori to sane default values based on our experience, but better ones can almost certainly be found.

The optimization problem can be solved using the `cvxpy` library, which is a Python library for convex optimization. The library provides an efficient interface for solving convex optimization problems, and it's widely used in the quantitative finance community. We encourage you to check out library online.

We implement the portfolio optimization problem in `hw4.py` in the function `run_portfolio_optimization`. The function calculates the optimized portfolio weights for a given set of return forecasts and residual returns (from which it calculates a simple covariance matrix forecast).

We run this function below to optimize the portfolio of residuals using the noisy oracle forecasting model with an IC of 0.10.

[ ]:
```
config.update({
    'RISK_AVERSION': 1.0,
    'MAX_LEVERAGE': 1.0,
```

```
    'MAX_WEIGHT': 0.05,
    'MAX_TURNOVER': 0.25,
    'TRANSACTION_COST': 0.0005,
    'BORROW_COST': 0.0001,
    'REBALANCING_PERIOD': 'W',
    'VERBOSE': False,
})

portfolio_weights, net_returns, performance_metrics =␣
 ↪run_portfolio_optimization(
    residual_returns,
    config,
    oracle_forecasted_residual_returns
)
```

Answer the following questions:

1. What do you notice about the performance?
2. Do you see any issues with this portfolio optimization approach? What would be a more accurate approach? (Hint: consider that the residuals are themselves portfolios)
3. Do you think the noisy oracle model of forecasting returns is useful? Why or why not? How could the noisy oracle model of forecasting returns be made more realistic?
4. Earlier, we explored one method of forecasting residual returns using the OU process signal. However, any other method of forecasting residual returns can be used in practice. In fact, the features used in the method are usually just as, if not more important, than the method itself. What kind of methods and features do you think would be useful for forecasting residual returns? Which would not be useful? Why? Please don't just list some generic machine learning methods; try to think about the specific context of forecasting residual returns and what features would be useful in that context.

Solution:

---

6 Question 6 (extra credit)

(2 points, extra credit)

In this homework, we've formed just one return forecast based on one signal: the Avellaneda-Lee s-score. In practice, you'll want to build return forecasting models which use many different signals (aka features). Moreover, you might have use different models for different assets or predict over a different horizon. In this question, we'll explore these avenues for improvement.

Do one of the following:

1. Make a new return forecast function which uses multiple features and/or a different horizon. Attempt to make changes which improve the IC on the data from the beginning of the sample period through the end of 2018. Does this improvement translate to a test set data composed of 2019 through the end of the sample period?

2. Examine the cross-section and time-variation of the ICes of the signal model we built earlier. Are there any patterns you can observe in the performance? Can you find a way to classify or predict which tickers have consistently high ICs?

Feel free to add any additional code to the `hw4.py` file and any code or markdown cells below to communicate your results. No need to paste functions you implement here; keep those in `hw4.py`.