IDS 435 - Final Project

Bermudan Options Pricing

Spring 2022

Table of Contents

Grading Table

Milestone	1	2	3
Grade Percentage	25%	30%	45%

Introduction

For the final project, you will study and solve a Financial engineering problem called Bermudan option pricing (BOP). The final project entails the following components:

- learning what the BOP problem is;
- understanding the formulation of a dynamic optimization problem representing BOP;
- executing a given python code to generate synthetic asset price data;
- implementing a popular algorithm, called Least Squares Monte Carlo (LSM), to solve BOP instances;
- extending LSM in different ways and implementing these extensions;
- benchmarking LSM extensions against vanilla LSM on BOP instances.

The specific version of BOP problem that we consider is the knock-out Bermudan call option pricing. Let's informally learn what are the components of this version of BOP. Let's

- **What is a call option?** Call options are *financial contracts* that give the option buyer the right but not the obligation to buy a stock, bond, commodity, or other asset or instrument at a specified price within a specific time period. The stock, bond, or commodity is called the underlying asset. A call buyer profits when the underlying asset increases in price. For more explanations and examples on call options, please check Investopedia webpage.

- **Example of a call option.** Suppose that Microsoft stock is trading at 108 dollars per share. You own 100 shares of the stock and want to generate an income above and beyond the stock's dividend. You also believe that shares are unlikely to rise above 115.00 dollars per share over the next month. You take a look at the call options for the following month and see that there's a 115.00 dollars call trading at 0.37 dollars per contract. So, you sell one call option and collect the 37 dollars premium (0.37×100 dollars shares), representing a roughly 4 percent annualized income. If the stock rises above 115.00 dollars, the option buyer will exercise the option, and you will have to deliver the 100 shares of stock at 115.00 dollars per share. You still generated a profit of 7.00 dollars per share, but you will have missed out on any upside above 115.00 dollars. If the stock doesn't rise above 115.00 dollars, you keep the shares and the 37 dollars in premium income. (The original source of example is Investopedia).

- **What is a Bermudan call option?** There are two main types or styles of options, American and European options. American options are exercisable at any point in time between the purchase date and the date of expiration. European options, however, are exercised only at the date of expiration. Bermudan options are a restricted form of the American option that allows for early exercise (i.e., before date of expiration) but only at predetermined points in time. For more explanations and examples on types of options, please check Investopedia webpage.

- **What is a knock-out option?** A knock-out option is an options contract that will become worthless if the investment reaches a specific price. In such a case, the options contract is "knocked out," and the investor will not receive a payoff. For more explanations and examples on call option, please check CFI webpage.

- **What does it mean to exercise a call option?** Exercise means to put into effect the right to buy or sell the underlying financial instrument specified in an options contract. When exercising a call option, the owner of the option purchases the underlying shares (or commodities, fixed interest securities, etc.) at the strike price from the option seller. For more explanations about the "exercising" options, please check Wikipedia) webpage.

Bermudan Options Pricing

Consider the pricing of a knock-out Bermudan call option with J assets over a finite time horizon. The option has T *exercise opportunities* over Y years; that is, exercise is possible at times $\{\tau, 2\tau, \ldots, T\tau\}$, where $\tau = Y/T$. The asset prices at time $t \in \mathcal{T} = \{0, 1, \ldots, T\}$ are $p_t = (p_{t,1}, p_{t,2}, \ldots, p_{t,J})$, where $p_{t,j}$ is the price of the j-th asset at this time. The option becomes *worthless* any time the maximum of the J asset prices exceeds a pre-specified *barrier price* p^{B}. We use the binary variable $y_t \in \{0,1\}$ to indicate if the option is knocked out at time t . It takes a value of one in this case and is zero otherwise. We can define random variable y_t recursively according to

$$y_t = \begin{cases} 1 & \text{if} \quad y_{t-1} = 1 \text{ or } \max_j\{p_{t,j}\} \geq p^{\mathrm{B}}, \\ 0 & \text{otherwise,} \end{cases}$$

for all $t > 0$ with the base case of

$$y_0 = \begin{cases} 1 & \text{if} \quad \max_j\{p_{0,j}\} \geq p^{\mathrm{B}}, \\ 0 & \text{otherwise.} \end{cases}$$

Price Model. We model the price stochastic process using an uncorrelated multi-asset geometric Brownian motion. Assume all assets share the same initial price $p^{\mathrm{I}} > 0$, that is, $p_{0,1} = p_{0,2} = \cdots = p_{0,J} = p^{\mathrm{I}}$. The price random variable at time $t > 0$ can be computed using the following formula for the geometric Brownian motion:

$$p_{t+1,j} = p_{t,j} \, \exp\left(\left(r - \frac{\sigma^2}{2}\right)\tau \, + \, \sigma\sqrt{\tau}Z_{t+1,j}\right), \qquad \forall j = 1, 2, \ldots, J.$$

where r is the risk-free interest rate, $\sigma > 0$ is the stock price volatility, and $Z_{t+1,j}$ is standard normal random variable. The general formulation of geometric Brownian motion and its examples can be found in $\S 3.2.3$ of Glasserman 2004 as well as this Towards Data Science post.

Markov Decision Process (MDP). MDPs provide a versatile mathematical framework for modeling sequential decision making problems (i.e., BOP) in the presence of stochasticity (i.e., prices). MDPs entail 4 components: (i) state, (ii) action, (iii) transition, and (iv) reward. We next define these components for BOP.

At time t, the MDP state is given by the vector $s_t = (p_{t,1}, p_{t,2}, \ldots, p_{t,J}, y_t)$ that encodes price information at the current time plus the knocked out binary variable y_t. The state vector s_t belongs to the set of all possible state, called state space, $\mathcal{S} = [0, p^{\mathrm{B}}]^J \times \{0,1\}$. The MDP action a_t is binary, with values of one and zero corresponding to "stop" and "continue," respectively. The MDP transition from one state to the other is given by the recursive definitions of $p_{t+1,j}$ and y_{t+1} in the previous subsection.

Stopping at stage t yields the MDP reward of $r_t(s_t, 0) = \gamma^t g(s_t)$, where the *discount factor* $\gamma = \exp(-r\tau)$ and the payoff function $g(\cdot) : \mathfrak{R}^{J+1} \mapsto \mathfrak{R}$ with respect to a pre-specified strike price p^{S} is

$$g(s_t) = \max\left\{\left(\max_{j=1,2,\ldots,J}\{p_{t,j}\} \; - \; p^{\mathrm{S}}\right), 0\right\}(1 - y_t).$$

Term $\max_j\{p_{t,j}\} - p^{\mathrm{S}}$ is the reward obtained by the stop decision and zero corresponds to the reward of continue decision. Therefore, the payoff of the option corresponds to that of a call option on the maximum of J non-dividend-paying assets with an up-and-out barrier.

The objective in BOP problem is to find the optimal stopping time. Formally, we want to find an exercise policy $\pi : \mathcal{S} \mapsto \{\text{stop}, \text{continue}\}$ that maximizes the discounted expected reward

$$\max_{\pi} \mathbb{E}_p^{\pi}\left[\sum_{t=1}^{T(\pi)} \gamma^t g(s_t) \,\middle|\, s_0 = (p_0, y_0)\right], \tag{PO}$$

where $T(\pi) := \operatorname{argmin}\{t \in \mathcal{T} : \pi(s_t) = \text{stop}\}$ is the stopping time under policy π. Note that in the above optimization problem s_{t+1} depends on the state and action at time t, that are, s_t and $\pi(s_t)$, respectively.

Synthetic Data Simulation

In this section, we learn how to simulate price trajectories for the BOP problem using geometric Brownian motion. Consider the following python code:

In [ ]:
```python
#------------------------------------------------------------------------------
# This is an implementation of multidimensional geometric Brownian motion
# that is inspired from the following codes:
#    https://towardsdatascience.com/how-to-simulate-financial-portfolios-with-py
#    https://github.com/Self-guided-Approximate-Linear-Programs/Self-guided-ALPs
#------------------------------------------------------------------------------

import numpy as np

def price_simulator(random_seed:int,
                    num_asset:int,
                    init_price:np.ndarray,
                    interest_rate:np.ndarray,
                    sigma:np.ndarray,
                    num_stages:int,
                    delta_t:float):
    """
    Geometric Brownian Motion
    ---
    Inputs:
        seed:           random seed for replicating the same simulation
        num_asset:      number of assets
        init_price:     initial stock prices
        interest_rate:  the risk-free interest rate, r.
        sigma:          volatility
        num_stages:     number of exercise opportunities
        delta_t:        elapsed time between two consecutive exercise opportunit

    Output:
        prices:         price of each asset over time
    """
    assert len(init_price) == num_asset
    assert len(interest_rate) == num_asset
    assert len(sigma) == num_asset

    np.random.seed(random_seed)
    prices                  = np.zeros([num_asset, int(num_stages)])
    prices[:, 0]            = init_price
    for i in range(1, int(num_stages)):
        drift               = (interest_rate - 0.5 * sigma**2) * delta_t
        Z                   = np.random.normal(0., 1., num_asset)
        diffusion           = Z * (np.sqrt(delta_t)) * sigma
        prices[:, i]        = prices[:, i-1]*np.exp(drift + diffusion)
    return prices
```

Generating price trajectories. We can use the above code to generate price trajectories for multiple assets. To see how this function works, we generate price sample paths for $J = 2$ assets over $Y = 3$ years with $T = 54$ exercise opportunities. We use $r = 5\%$ interest rate and $\sigma = 20$ stock price volatility for all assets. We employ the initial price $P^{\mathrm{I}} = 90$. We summarize these values in the following table:

Parameter	Description	Value
J	Number of assets	2
Y	Years of investment	3
T	Number of exercise opportunities	54
r	Interest rate for all assets	5%
σ	Stock price volatility for all assets	20%
P^{I}	Initial price for all assets	90

We are ready now to generate price trajectories and depict them. For plotting the prices, we use the following code:

In [ ]:

```
def plot_prices(prices:np.ndarray,num_asset:int,num_stages:int):
    fig, ax  = plt.subplots(figsize=(6, 4),dpi=100)
    for i in range(num_asset):
        ax.plot(prices[i,:],lw=2,alpha=.5,label='Asset '+str(1+i),marker='.')

    min_p,max_p = float(np.min(prices)),float(np.max(prices))
    ax.set_xticks(np.arange(0,num_stages+1,4))
    ax.set_yticks(np.round(np.arange(min_p,max_p+1,int((max_p - min_p)/10))))
    ax.set_xlabel(r'stage ($t$)')
    ax.set_ylabel(r'price ($p_{t,j}$)')
    plt.legend()
    plt.show()
    return True
```

Generating prices and plotting them.

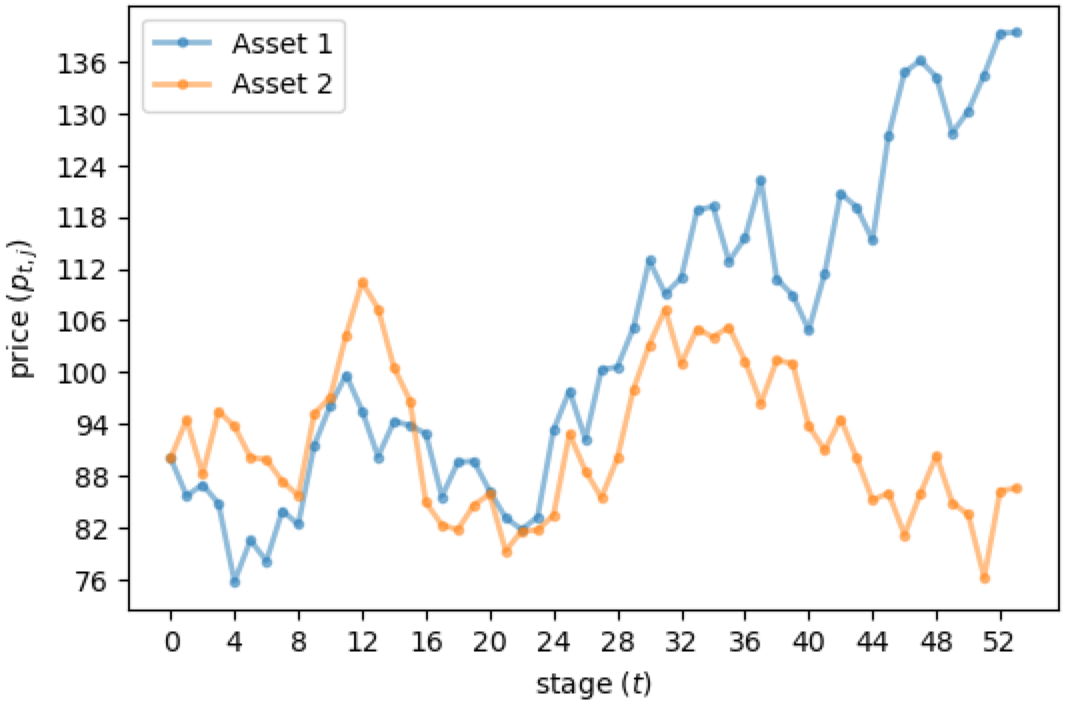
In [ ]:

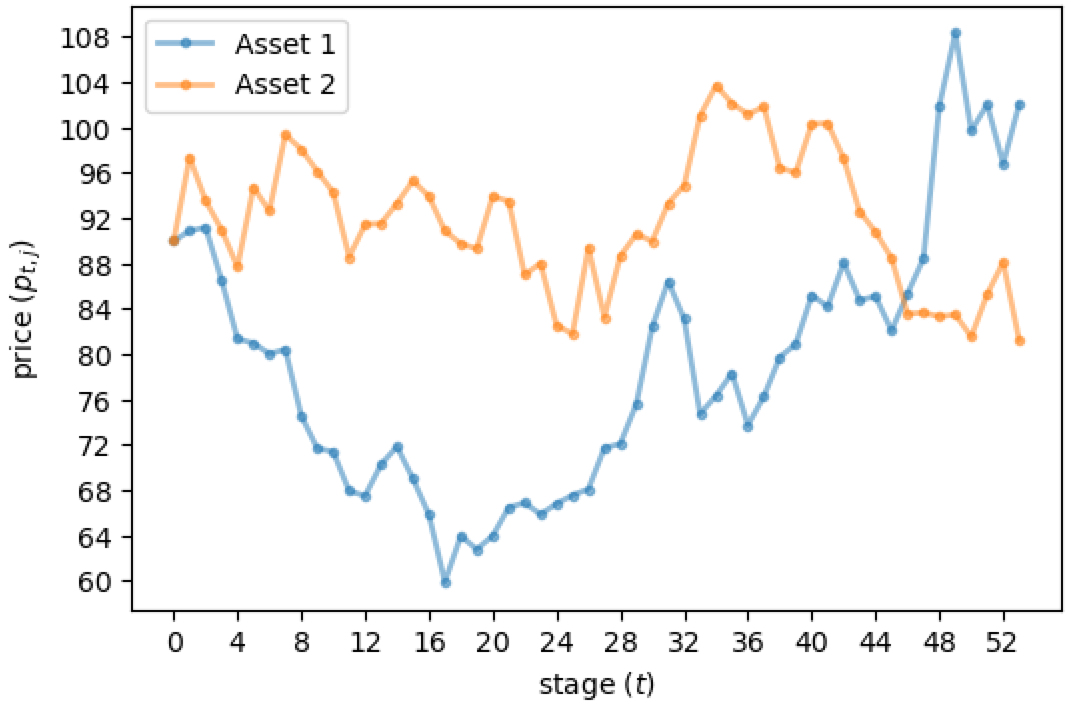
```
import matplotlib.pyplot as plt

if __name__== "__main__":
    time_horizon     = 3    # number of years, Y.
    num_stages       = 54   # number of exercise opportunities, T.
    delta_t          = time_horizon/num_stages   # elapsed time, tau
    num_asset        = 2
    init_price       = 90
    prices_realization_1    = price_simulator(
                        random_seed          = 123,
                        num_asset            = num_asset,
                        init_price           = np.array([init_price,init_price]),
                        interest_rate        = np.array([0.05,0.05]),
                        sigma                = np.array([0.20,0.20]),
                        num_stages           = num_stages,
                        delta_t              = delta_t,
                    )
    plot_prices(prices_realization_1,num_asset,num_stages)

    prices_realization_2    = price_simulator(
                        random_seed          = 321,
                        num_asset            = num_asset,
                        init_price           = np.array([init_price,init_price]),
                        interest_rate        = np.array([0.05,0.05]),
                        sigma                = np.array([0.20,0.20]),
                        num_stages           = num_stages,
                        delta_t              = delta_t,
                    )


    plot_prices(prices_realization_2,num_asset,num_stages)
```





How to interpret the above plots? Consider the top plot. It shows a price trajectory per asset, where each trajectory entails 54 price values shown via circular markers. We have the blue (orange) trajectory corresponding to the first (second) asset. Since price values are stochastic, it is possible to see price trajectories other than those shown in the top figure. The bottom figure shows another pair of price trajectories for the same assets where the realization of the prices are different from the realization in the top plot. Both plots show possible price trajectories that an investor might encounter while investing.

Milestone 1

Your first milestone entails two following steps:

- Write a one-page problem statement that concisely states the BOP problem. Please first read §Bermudan Options Pricing carefully and then include the following information in your one-page write-up.
    - Write a paragraph to present the problem of Bermudan options pricing;
    - Write a paragraph to present the components of the MDP modeling the BOP problem (do not need to explain how prices are simulated);
    - Write the third paragraph to explain the exercise policy optimization problem. Please describe what this optimization problem precisely does by describing its objective function, decision variables, and constraints

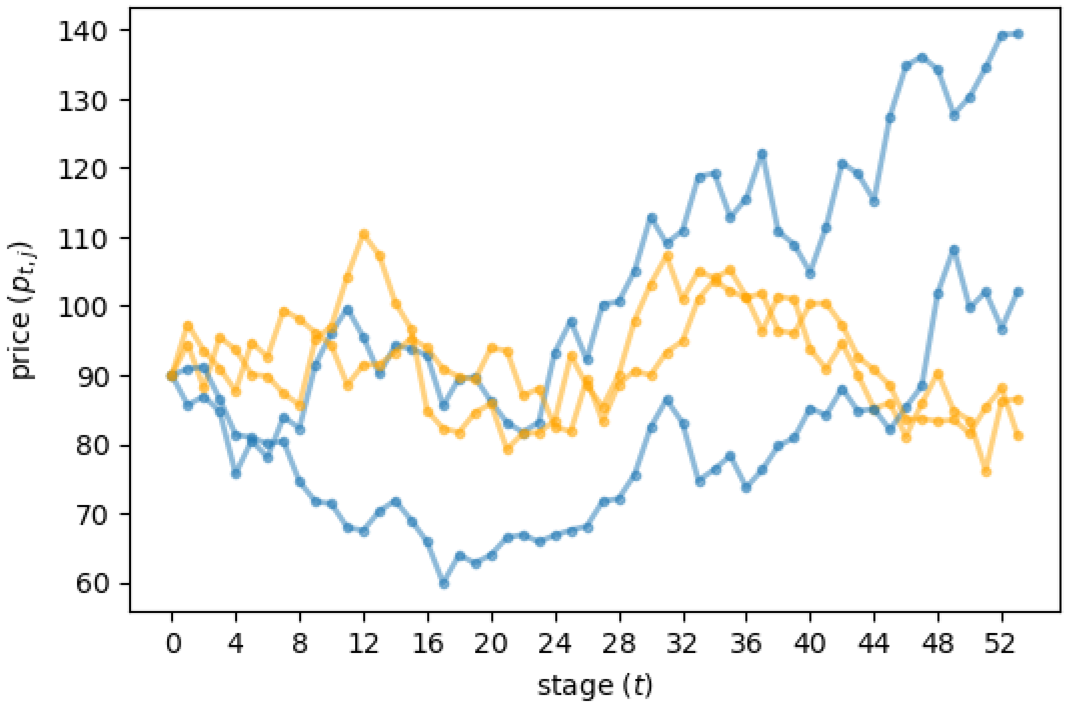
    Note that the purpose of the above writing is to ensure that you got familiar with the BOP problem, understood the MDP components (i.e., state, action, reward, and transitions), and thought about the meaning of optimizing policy.

- Extend function `price_simulator` presented in §Synthetic Data Simulation to `price_simulator_multiple_trajectories` that generates several price trajectories at the same time. Specifically, `price_simulator_multiple_trajectories` should receive the additional input `num_trajectories` compared to function `price_simulator` and returns several realizations of price trajectories. You should also extend the plotting function `plot_prices` to depict all trajectories obtained from `price_simulator_multiple_trajectories` in one figure. Below is an example of how we expect your code to work.

Your function `price_simulator_multiple_trajectories` should be called similar to the following code:

```
In [ ]:
prices               = price_simulator_multiple_trajectories(
                           random_seed             = 123,
                           num_asset               = num_asset,
                           init_price              = np.array([init_price,init_price]),
                           interest_rate           = np.array([0.05,0.05]),
                           sigma                   = np.array([0.20,0.20]),
                           num_stages              = num_stages,
                           delta_t                 = delta_t,
                           num_trajectories        = 2
                       )
```

If you use your extended plotting function to depict the `prices` generated above, then you should see a plot similar to following figure:



Submission format.

Turn in a *Jupyter Notebook* file as well as *its associated HTML* that entails both your one-page write-up and the code for price simulation. The due date is Monday, April 4th, at 7:00 PM CDT.

Your files must be submitted to the Blackboard. *No late submission is allowed.*

Least Squares Monte Carlo

A major component in this project is to learn and implement Least Squares Monte Carlo (LSM), a popular algorithm for pricing financial (i.e., Bermudan) and real options.

Policy optimization.

Recall the optimization problem PO from §Bermudan Options Pricing that is repeated below:

$$\max_{\pi} \mathbb{E}_p^{\pi}\left[\sum_{t=1}^{T(\pi)} \gamma^t g(s_t) \,\Big|\, s_0 = (p_0, y_0)\right]. \tag{PO}$$

PO finds an optimal policy that maximizes the expected discounted payoff of stopping. Each policy $\pi = (\pi_0, \pi_1, \ldots, \pi_T)$ is a collection of (T+1) functions π_t, where $\pi_t(\cdot)$ receives state vector s as input and tell us to either stop at time t or continue. Therefore, PO has these functions as its decision variables. It is unclear how to solve it given the tools we learned so far in IDS 435! In the subsequent subsections, we explain how an optimal policy, which is an optimal solution of PO, can be computed without directly solving PO using tools we already learned in IDS 435.

Value and continuation function.

For a policy $\pi = (\pi_1, \pi_2, \ldots, \pi_T)$, define $T_\pi(t) = \min\{t' \geq t : \pi_{t'}(s_{t'}) = \text{stop}\}$ as the earliest time after t that π stops. The "value" of this policy when we start from state s at time t is defined as:

$$V_t^{\pi}(s) := \mathbb{E}\left[\gamma^{T_\pi(t)-t}\, g(s_{T_\pi(t)}) \mid s_t = s\right], \tag{Value Function}$$

where term $s_{T_\pi(t)}$ is the state at the stopping time $T_\pi(t)$, $g(s_{T_\pi(t)})$ is the payoff collected at state $s_{T_\pi(t)}$, $T_\pi(t) - t$ is the number of time periods between the current time t and the stopping time $T_\pi(t)$, and $\gamma^{T_\pi(t)-t}$ is the discount factor. The conditional expectation in the definition of *value function* V_t^{π} encodes the probability of starting from state $s_t = s$ at the current time t and arriving to the state $s_{T_\pi(t)}$ at time $T_\pi(t) \geq t$. Note that this probability solely depend on the evolution of the prices since state vector only relies on the price information. Utilizing the definition of *value function* $V^{\pi} := (V_1^{\pi}, \ldots, V_T^{\pi})$, our goal of solving PO to find an optimal policy can be restated as finding an optimal policy π^* that maximizes the value function $V_t^{\pi}(s)$ for all times t and all states s. We refer to the value function associated with the optimal policy π^* solving PO as the "optimal value function".

Value function recursion.

Denote the optimal value function by $V^* = (V_0^*, V_1^*, \ldots, V_T^*)$. We can compute it using the following recursion that connects V_t^* to V_{t+1}^*:

$$V_t^*(s) := \begin{cases} \max\left\{g(s),\ \gamma\mathbb{E}\left[V_{t+1}^*(s_{t+1}) \mid s_t = s\right]\right\}, & \text{if} \quad t \leq T-1 \\ g(s) & \text{if} \quad t = T. \end{cases}$$

We interpret the above recursion as follows.

- Assume at time $t \leq T-1$ you faced state vector s. Then Payoff $g(s)$ tells you the value you collect if you stop at the current time t and the value of $C_t^*(s) := \gamma\mathbb{E}\left[V_{t+1}^*(s_{t+1}) \mid s_t = s\right]$ tells you the optimal discounted expected value of continuing. Among the two options of stopping or continuing, the optimal value function should pick the one with highest value. Therefore, $V_t^*(s)$ is the maximum of $g(s)$ and $C_t^*(s)$.
- For any state vector s at time $t = T$, there is no option other than stopping. Hence, the optimal value at $t = T$ is given by the payoff of the state encounter at $t = T$.

We thus call $C^* = (C_0^*, C_1^*, \ldots, C_T^*)$ the optimal *continuation function*.

Why continuation function is useful?

If we access the optimal continuation function, then the optimal policy can be compute via the following simple rule without a need to solve PO:

$$\pi_t^*(s) := \begin{cases} \text{continue}, & \text{if} \quad t \leq T-1, \text{ and } g(s) \leq C_t^*(s), \\ \text{stop}, & \text{otherwise.} \end{cases}$$

One could argue that C^* is not known apriori to compute π^* using the above simple rule. Sure, but we can train a model for each C_t^* using an appropriate dataset and construct the optimal policy using the above rule and approximate continuation function values.

Continuation Function Approximation(CFA).

We can construct a recursion for the optimal continuation function similar to the value function as follows:

$$C_t^*(s) := \begin{cases} \gamma\mathbb{E}\left[\max\left\{g(s_{t+1}),\ C_{t+1}^*(s_{t+1})\right\} \,\middle|\, s_t = s\right], & \text{if} \quad t \leq T-1 \\ 0 & \text{if} \quad t = T. \end{cases}$$

Again, while the above recursion allows us to compute the optimal policy directly without solving PO, we do not know what is the optimal continuation function and thus approximation is needed.

Let's leverage what we learned in IDS 435. Throughout the course, we learned how to construct predictive models, i.e., linear regression. Using a similar idea, we can approximate the optimal continuation function. At every time t and for the $(J+1)$-dimensional state vector $s_t = (p_t, y_t)$, we define the following approximation:

$$C_t^*(s_t) \approx (1-y_t)\Phi_t(s_t; w), \qquad \Phi_t(s_t; w) = \sum_{b=1}^{B} w_{t,b}\phi_{t,b}(s_t).$$

Here, $(1-y_t)\Phi_t(s_t; w)$ is the CFA. Note that if $(1-y_t) = 0$, then the option is knocked-out and thus $C_t^*(s_t) = 0$. Otherwise, $(1-y_t) = 1$ and thus our approximation becomes $C_t^*(s_t) \approx \Phi_t(s_t; w)$. Model $\Phi_t(s_t; w)$ entails B features $\{\phi_{t,b}(s_t) : b = 1, 2, \ldots, B\}$ of the state vector s_t. Let us provide some examples of these features.

- *Linear CFA*. Fix time $t = 1, 2, \ldots, T$. If we set $B = J$ and use linear features $\phi_{t,b}(s_t) = p_{t,b}$ for all $b = 1, 2, \ldots, J$, then we get the following linear approximation:

$$\Phi_t(s_t; w) = \sum_{b=1}^{J} w_{t,b}\, p_{t,b}$$

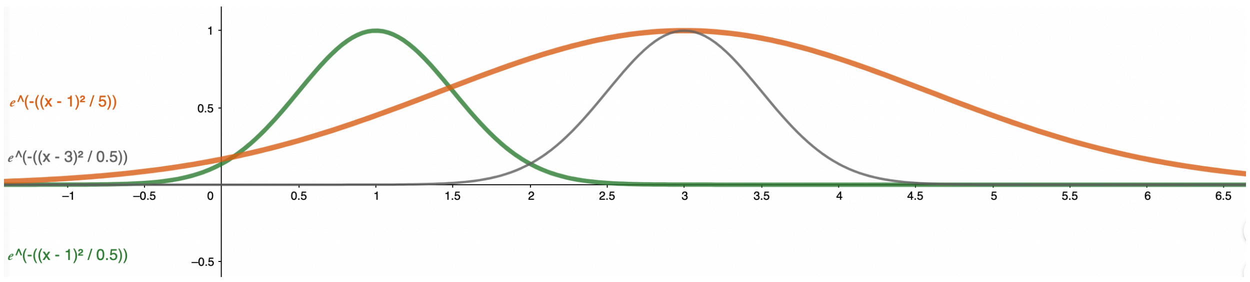
- *DFM CFA*. A popular choice of features $\phi_{t,b}(\cdot)$ for BOP problem is presented in §4.3.1 of Desai et al 2012 (abbreviated DFM). This choice entails $B = J + 2$ features as follows:

$$\begin{aligned} \phi_{t,1}(s_t) &= 1, & \forall t &= 0, 1, \ldots, T \\ \phi_{t,2}(s_t) &= g(s_t), & \forall t &= 0, 1, \ldots, T \\ \phi_{t,2+b}(s_t) &= p_{t,b}, & \forall b &= 1, 2, \ldots, B; \forall t = 0, 1, \ldots, T. \end{aligned}$$

- *Kernel CFA*. Kernel trick is popular approach to construct non-linear models (i.e., see kernel regression, [kernel classification]). A kernel is a function that measures the similarity between two vectors. Let say s_t and $\hat{s}_t$ are two state vectors at time t, then kernel K for pair $(s_t, \hat{s}_t)$ is the non-negative value $K(s_t, \hat{s}_t) \geq 0$ that measure similarity between s_t and $\hat{s}_t$. Radial basis function kernel $K(s_t, \hat{s}_t) = \exp(-\|s_t - \hat{s}_t\|_2^2 / 2\rho)$ is a widely-used kernel function. We call ρ bandwidth of the kernel and $\hat{s}_t$ the centroid of the kernel. Given a kernel K and a set of prices $\{\hat{s}_{t,b} : b = 1, 2, \ldots, B\}$, we can construct CFA using the following features:

$$\phi_{t,b}(s_t) = K(s_t, \hat{s}_{t,b}),$$

Below is a picture of a radial basis function kernel with different centroids and bandwidths.



For a given CFA class (i.e., linear CFA), our goal is to optimize coefficients $\{w_{t,b} : b = 1, 2, \ldots, B\}$ to make $\Phi_t(\cdot; w)$ close to V_t^* as much as possible for every $t = 1, 2, \ldots, T$. LSM is an algorithm that uses iterative regression to compute coefficients $\{w_{t,b} : b = 1, 2, \ldots, B\}$. Since we need data to do regression, let's talk about data before discussing LSM.

Training Data for Fitting CFA.

Assume you have generated M price trajectories using your code from milestone 1. These trajectories are given by the set $\{(s_0^m, s_1^m, \ldots, s_T^m) : m = 1, 2, \ldots, M\}$. For trajectory m, where $s_t^m = (p_t^m, y_t^m)$. Each price vector $p_t^m = (p_{t,1}^m, p_{t,2}^m, \ldots, p_{t,J}^m)$ encodes the prices of J assets at time t, and y_t^m shows if the option is knocked-out at time t or not. One can thus think of training data as a 3D matrix of dimension $J \times T \times M$, where J is the number of assets, T is the number of exercise opportunities, and M is the number training trajectories.

Define payoff M-dimensional vector g_t as well as $M \times B$ dimensional *feature matrix* F_t as follows:

$$g_t = \begin{bmatrix} g(s_t^1), \\ g(s_t^2), \\ \vdots \\ g(s_t^M) \end{bmatrix}, \qquad F_t = (1 - y_t) \begin{bmatrix} \phi_{t,1}(s_t^1), & \phi_{t,2}(s_t^1), & \ldots, & \phi_{t,B}(s_t^1) \\ \phi_{t,1}(s_t^2), & \phi_{t,2}(s_t^2), & \ldots, & \phi_{t,B}(s_t^2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{t,1}(s_t^M), & \phi_{t,2}(s_t^M), & \ldots, & \phi_{t,B}(s_t^M) \end{bmatrix}_{M \times B}$$

Note that for a weight vector $w \in \Re^B$, we have $F_t w$ equals

$$F_t w = \begin{bmatrix} (1 - y_t) \sum_{b=1}^B w_b \phi_{t,b}(s_t^1) \\ (1 - y_t) \sum_{b=1}^B w_b \phi_{t,b}(s_t^2) \\ \vdots \\ (1 - y_t) \sum_{b=1}^B w_b \phi_{t,b}(s_t^M) \end{bmatrix}_{M \times 1} = \begin{bmatrix} (1 - y_t)\Phi(s_t^1; w) \\ (1 - y_t)\Phi(s_t^2; w) \\ \vdots \\ (1 - y_t)\Phi(s_t^M; w) \end{bmatrix}_{M \times B} .$$

Thus, for a given $w \in \Re^B$, $F_t w$ is an M dimensional vector that encodes the value of approximation model $(1 - y_t)\Phi(s_t^m; w)$ for all $m = 1, 2, \ldots, M$.

LSM Algorithm for Fitting CFA.

Algorithm LSM relies on two components that we already discussed: (i) CFA

$$C_t^*(s) \approx (1 - y_t) \left(\sum_{b=1}^B w_{t,b} \phi_{t,b}(s) \right),$$

and (ii) the following recursion

$$C_t^*(s) := \begin{cases} \gamma \mathbb{E}\left[\max\left\{g(s_{t+1}),\ C_{t+1}^*(s_{t+1})\right\} \,\middle|\, s_t = s\right], & \text{if} \quad t \le T-1 \\ 0 & \text{if} \quad t = T. \end{cases}$$

Putting together these components, LSM is given by the following steps:

- Fix model Φ, and receive M training price trajectories $\{(s_0^m, s_1^m, \ldots, s_T^m) : m = 1, 2, \ldots, M\}$;
- At the terminal stage $t = T$, set $\hat{w}_T = 0$ since the options expire at the the terminal stage T and the continuation value is zero.
- For $t = T-1, T-2, \ldots, 0$ do:
    - Compute the continuation vector $c_t = \gamma \max\{g_{t+1}, F_{t+1}\hat{w}_{t+1}\}$
    - Set $\hat{w}_t$ to the optimal solution of the following least squares problem $\min_w \left\{\|c_t \ - \ F_t w\|_2^2\right\}$.
- return optimal weights $\{\hat{w}_t : t = 0, 1, 2 \ldots, T\}$.

Policy Simulation.

Assume you executed LSM algorithm and got the optimal CFA weights $\{\hat{w}_t : t = 0, 1, 2 \ldots, T\}$. How can you get the policy and compute its payoff? We need to perform the following steps, which we refer to as policy simulation:

- Generate a new set of price trajectories. Call them testing trajectories.
- Compute the earliest stopping time on each price trajectory in the testing set using CFA computed from LSM.
- Compute the average payoff across these trajectories using the earliest stopping time of each trajectory.

We formalize the above procedure in below. Let $\{(s_0^n, s_1^n, \ldots, s_T^n) : n = 1, 2, \ldots, N\}$ be the set of N testing trajectories. We can compute the policy associated with the CFA with weights $\{\hat{w}_t : t = 0, 1, 2 \ldots, T\}$ as follows:

- For each trajectory $n = 1, 2, \ldots, N$, do:
    - For time $t = 0, 1, \ldots, T-1$, do:
        - Compute $c_{t,n} := \gamma \max\left\{g(s_{t+1}^n), (1 - y_{t+1})\left(\sum_{b=1}^B \hat{w}_{t+1,b}\, \phi_{t+1,b}(s_{t+1}^n)\right)\right\}$.
        - If $c_{t,n} < g(s_t^n)$, then let $r_n := \gamma^t g(s_t^n)$ and break the inner loop.
    - If r_n is not assigned yet, then set $r_n := \gamma^T g(s_T^n)$.
- Return average payoff $\frac{1}{N}\sum_{n=1}^N r_n$.

Milestone 2

Please complete the followings:

1. Please read §Least Squares Monte Carlo until Monday, Apr 11, and make sure that you learned LSM algorithm and the concept of CFA.

2. Write a Python code that entails three functions:

   - Your code from Milestone 1 for generating price trajectories;
   - A new function that implements the LSM algorithm for computing weights of CFA (to solve the least squares optimization problem in the LSM algorithm, you can use "np.linalg.lstsq");
   - A new function for simulating policy using CFA that is computed by LSM;

     Note that your LSM implementation should be general to handle different CFA classes, including, linear, DFM, and kernel CFAs), among others that you will work on the following milestones.

3. Generate two separate training and testing price trajectories with the sizes of $M = 3000$ and $N = 3000$ for the following instance of the BOP problem:

Parameter	Description	Value
J	Number of assets	4
Y	Years of investment	3
T	Number of exercise opportunities	54
r	Interest rate for all assets	5%
σ	Stock price volatility for all assets	20%
P^{I}	Initial price for all assets	90
P^{B}	Barrier price	170
P^{S}	Strike price	100

1. Use linear CFA model in your LSM code to compute the optimal weights $\{\hat{w}_t : t = 0, 1, 2 \ldots, T\}$. Then, compute the average payoff of CFA associated with these optimal weights via the policy simulation procedure. Report your average payoff value (which should be greater than 33.011).

2. Repeat Step (4) above using DFM CFA model. Report your average payoff value and compare it against the payoff value from the linear model.

3. Consider kernel CFA. At every time t, a possible choice for the centroids of radial basis function kernels is to set them to the percentile of the training price values. Repeat Step (4) above using using radial basis function kernels and these centroids. Report your average payoff value and compare it with the values of linear and DFM CFA models. Comment on how do you choose bandwidth parameter ρ for radial basis function kernels.

Milestone 3

You have already implemented LSM for learning CFA. In the last milestone, you will figure out how to tune LSM to deliver higher payoffs. To this end, you will explore how the following components impact the quality of LSM policy:

- Adding regularizer to the least-squares problem of LSM;
- Cross-validating the LSM's regularization parameter;
- Cross-validating the bandwidth of kernels in kernel CFA;
- Cross-validating the number of centroids in the kernel CFA;

Throughout this milestone, we fix the following parameters:

Parameter	Description	Value
Y	Years of investment	3
T	Number of exercise opportunities	54
r	Interest rate for all assets	5%
σ	Stock price volatility for all assets	20%
P^{B}	Barrier price	170
P^{S}	Strike price	100
M	Number of training sample paths	10,000
N	Number of testing sample paths	10,000
K	Number of validation sample paths	10,000

Step 1 (Fix LSM Implementations).

Based on my quick evaluation of Milestone 2 submissions, most teams obtained strange payoff values from the LSM policy. Therefore, there are errors in the current implementations. As the first step in Milestone 3, you should fix your LSM implementation. To help you with this, I will give you feedback on your Milestone 2 submissions. Additionally, you will use the following table to fix your code:

Plain LSM

Instance	Number of assets (J)	Initial Price P^{I}	Lower Bound	Upper Bound	Linear CFA	DFM CFA	Kernel CFA
1	4	90	32.00	34.98			
2	4	100	40.50	43.85			
3	4	110	46.50	50.18			
4	8	90	43.12	34.98			
5	8	100	49.00	43.85			

Instance	Number of assets (J)	Initial Price P^{I}	Lower Bound	Upper Bound	Linear CFA	DFM CFA	Kernel CFA
6	8	110	52.11	55.06			

This table has 6 instances of the problem. In rows, we varied the initial price and the number of assets. The table provides you with the **lower** and **upper** bounds on the optimal payoff. As the first step in Milestone 3, complete the above table by reporting the average payoff of LSM policy based on linear, DFM, and kernel CFAs you had in Milestone 2. Note that you need to just run your code from Milestone 2 for the above 6 instances. For each instance, the value that you obtain from your implementation must be between the lower and the upper bounds provided in the table. If this is not the case, then your code is off! Use the above table and fix your code.

Step 2 (Regularization).

In your current LSM code, you solve the least-square problem $\min_w \left\{\|c_t - F_t w\|_2^2\right\}$ at every stage t. In Step 2, You will answer the question of: "can LSM performance be improved by adding a regularizer term to LSM least-square problems?" Modify your code to be able to perform λ-regularized least-square problem $\min_w \left\{\|c_t - F_t w\|_2^2 + \lambda\|w\|_2\right\}$, where $\lambda \geq 0$ is a constant. Run your code for $\lambda := \frac{1}{\sqrt{B}}$ and complete the following table:

Regularized LSM with $\lambda = \frac{1}{\sqrt{B}}$

Instance	Number of assets (J)	Initial Price P^{I}	Lower Bound	Upper Bound	Linear CFA	DFM CFA	Kernel CFA
1	4	90	32.00	34.98			
2	4	100	40.50	43.85			
3	4	110	46.50	50.18			
4	8	90	43.12	34.98			
5	8	100	49.00	43.85			
6	8	110	52.11	55.06			

Step 3 (Cross Validation of Regularization Parameter).

The choice of $\lambda := \frac{1}{\sqrt{B}}$ in Step 2 might not be the best choice! Let's do cross validation and pick the optimal λ value. Your current code has training and testing phases. You are going to add a validation phase. Generate $K = 5,000$ sample paths using your price generation code from Milestone 1. Then, for each instance and for linear, DFM, and kernel CFA models, try different values $\lambda \in \{2^{-v} : v = -4, -3, \ldots, -1, 1, \ldots, 4\}$ and find the optimal λ^* with the highest validation payoff. In other words, perform the following steps that are known as **cross-validation**:

- train your CFA using λ-regularized LSM and M training sample paths;
- compute the average payoff of LSM policy on K validation price trajectories using your policy simulation code from Milestone 2;
- set λ^* to the λ among candidates $\{2^{-v} : v = -4, -3, \ldots, -1, 1, \ldots, 4\}$ that leads to the highest validation payoff;
- re-fit your CFA via λ^*-regularized LSM and report the testing payoff for the optimal choice of λ^* in the following table:

Regularized LSM with optimal λ^*

Instance	Number of assets (J)	Initial Price P^{I}	Lower Bound	Upper Bound	Linear CFA	DFM CFA	Kernel CFA	λ^* for linear CFA	λ^* for DFM CFA	λ^* for kernel CFA
1	4	90	32.00	34.98						
2	4	100	40.50	43.85						
3	4	110	46.50	50.18						
4	8	90	43.12	34.98						
5	8	100	49.00	43.85						
6	8	110	52.11	55.06						

Step 4 (Number of Centroids).

We next try a heuristic approach to tune the number of centroids B used in the kernel CFA model. As we discussed in Milestone 2, kernel CFA is identified by the values of centroids $\{\hat{s}_{t,b} : b = 1, 2, \ldots, B\}$. Set $\hat{s}_{t,b}$ to the P-th percentile of the training prices at stage t, where

- Case 1: $B = 3$ and $P = (b * 25)\%$ for $b = 1, 2, 3$;
- Case 2: $B = 9$ and $P = (b * 10)\%$ for $b = 1, 2, \ldots, 9$.
- Case 3: $B = 19$ and $P = (b * 5)\%$ for $b = 1, 2, \ldots, 19$.

For each of these cases, you still need to select the value for ρ! Run your cross-validation and LSM code without regularization to find the best value of $\rho \in \{2^{-v} : v = -4, -3, \ldots, -1, 1, \ldots, 4\}$. Then, in the following table, for each of the above cases, report the average payoff on the testing set for the optimal ρ^*.

Plain LSM with kernel CFA and optimal ρ^*

Instance	Number of assets (J)	Initial Price P^{I}	Lower Bound	Upper Bound	Kernel CFA for $B=25$	Kernel CFA for $B=10$	Kernel CFA for $B=5$	Optimal ρ^* for $B=25$	Optimal ρ^* for $B=10$	Optimal ρ^* for $B=5$
1	4	90	32.00	34.98						
2	4	100	40.50	43.85						
3	4	110	46.50	50.18						
4	8	90	43.12	34.98						
5	8	100	49.00	43.85						
6	8	110	52.11	55.06						

Step 5 (Tuning Kernel CFA for LSM).

For each instance in the table of Step 4, pick the kernel CFA model with the highest test payoff. For example, for instance 1, you might realize that the kernel CFA with $B = 5$ and $\rho^* = 2^{-3}$ are the best choices and for instance 2, the kernel CFA with $B = 19$ and $\rho^* = 2^2$ are the best values. For these optimal choices of B^* and ρ^*, rerun your cross-validation code for λ-regularized LSM model and tune λ. Then, complete the following table:

Regularized LSM for kernel CFA with optimal B^*, optimal ρ^*, and optimal λ^*

Instance	Number of assets (J)	Initial Price P^{I}	Lower Bound	Upper Bound	Kernel CFA with optimal B^*	Optimal B^*	Optimal ρ^*	Optimal λ^*
1	4	90	32.00	34.98				
2	4	100	40.50	43.85				
3	4	110	46.50	50.18				
4	8	90	43.12	34.98				
5	8	100	49.00	43.85				
6	8	110	52.11	55.06				

What to turn in?

You will turn in an 8-page write-up that **must be in PDF** format and a Jupyter notebook that has your entire code. I will grade your write-up first, and if needed, I will check your code from the

Jupyter notebook. Thus, please turn in a well-written PDF write-up. Please format it according to the following points:

- Page 1: write the problem definition using what you turned in in Milestone 1;
- Page 2: report the results obtained in Step 1 of Milestone 3, include the completed table in Step 1 of Milestone 3, and explain the results obtained in the table.
- Page 3 (4,5, and 6): report the results obtained in Step 2 (3,4, and 5) of Milestone 3, include the completed table in Step 2 (3,4, and 5) of Milestone 3, and explain the results obtained in the table.
- Pages 7 and 8: aadd a discussion on the results you obtained in Steps (2)-(5) of Milestone 3. Please write these pages such that if I ready Pages 7 and 8, I should understand what version of LSM with what CFA model leads to the best policy payoff and which combination is the fastest. To be precise, please add the following information on these pages:
  - compare and contrast the performance linear, DFM, and kernel CFA models; which one leads to the best policy? which one is the worst? please elaborate;
  - compare the runtime of LSM with and without regularization;
  - explain how the runtime and policy performance of LSM with kernel CFA varies when we change B between values $3, 9,$ and 19 kernel;
  - between tuning of λ, ρ, and B, which one results in the most improvement in the policy performance?

References

- Desai, Vijay V., Vivek F. Farias, and Ciamac C. Moallemi. "Pathwise optimization for optimal stopping problems." Management Science 58, no. 12 (2012): 2292-2308.
- Nadarajah, Selvaprabu, François Margot, and Nicola Secomandi. "Comparison of least squares Monte Carlo methods with applications to energy real options." European Journal of Operational Research 256, no. 1 (2017): 196-204.
- Glasserman, Paul. Monte Carlo methods in financial engineering. Vol. 53. New York: springer, 2004.
- D'Elia, Riccardo. How to simulate financial portfolios with Python: An application of Multidimensional Geometric Brownian Motion. Link to the blog post.
