Study of a buy-and-hold investment using Monte Carlo approach

David CICORIA

January 4, 2019

Abstract

Monte Carlo approach is used with geometric Brownian motion (GBM) to simulate a buy-and-hold investment for a period of 10 years. Model properties such as the annual rate of return μ, and annual volatility σ are determined from historical price. Different properties are then plotted or assessed to characterize the investment after 10 years of holding.

1. Historical evaluation

The data consists of a univariate data series of price for around 13 years. The first step before running Monte Carlo simulations is to estimate the important parameters: annual rate of return $\mu = \mu_{annu}$, and annual volatility $\sigma = \sigma_{annu}$. In this study, we use PYTHON language. Starting from the data in xlsx (Excel) format, finding the parameters from historical data requires several operations that are described below.

First, the different modules necessary for data processing and plotting are imported in the script. A 'DataFrame' is created from the univariate series extracted from the Excel file. Different operations are carried out, such as parsing the dates, checking if any values are NaN, and so on. Finally, two new columns are added containing the return of the ETF, and its logarithmic counterpart. Both returns can actually be used for further calculation. However, in this study we consider the regular daily return of the ETF defined as $R_{ETF}(day)$, such that

$$R_{ETF}(day+1) = \frac{P_{ETF}(day+1) - P_{ETF}(day)}{P_{ETF}(day)} \quad , \tag{1}$$

where $P_{ETF}(day)$ corresponds to the price of the ETF on a particular day. The following piece of code shows all of the operations realized at that stage.

```
# Import all modules used in this script
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from math import sqrt

#Import all functions (models) from Monte_Carlo_GBM.py
from Monte_Carlo_GBM import Monte_Carlo_SDE_GBM
from Monte_Carlo_GBM import Alternative_Monte_Carlo_GBM
from Monte_Carlo_GBM import Analytic_Solution_GBM
from Monte_Carlo_GBM import evaluate_algorithms
```

```python
#Import all functions (post-proc) from Post_processing.py
from Post_processing import plot_price_with_time
from Post_processing import plot_histogram
from Post_processing import plot_histogram_together
from Post_processing import plot_kde
from Post_processing import plot_kde_together
from Post_processing import print_percentiles
from Post_processing import show_stats


# Import data from Excel as a DataFrame for python; date is parsed and converted to
↪  python date format
ETF_data = pd.read_excel('Quant Dev_CandidateExercise_Price_Rev_201811.xlsx',
↪  index_col=0, parse_dates = ['date'])

# Date for indexation of dataframe
ETF_data.index.name = 'date'
# Check whether there is any NaN values in the dataset
print('Is there any NaN values in the dataset :', ETF_data.isnull().values.any())
# Look at the values of the dataframe
values = ETF_data.values
# Ensure all data is float
values = values.astype('float32')

# Add column for the return of the ETF
ETF_data['return'] = (ETF_data['close'].pct_change()).shift(-1)
# Add column for the logarithmic return of the ETF
ETF_data['ln_return'] = (np.log(ETF_data['close'].shift(-1) / ETF_data['close']))
```

Next, we want to calculate the number of trading days per year. To do so, we resample our data series on a yearly basis. The final year (2018) is removed as it is not yet finished, and would deteriorate our estimation. The number of trading days is then selected by considering the average number of trading days over the years. We find that the number of trading days is **260**. The code below highlights all of these steps.

```python
# Counting number of days in a year
count_yearly_days = ETF_data['close'].resample("Y").count()
# Removing the last row of the dataset because 2018 is not finished yet
count_yearly_days = count_yearly_days[:-1]
# Averaging the number of days for each year to get the average number of trading
↪  days and change it as an integer
trading_days_per_year = int(count_yearly_days.mean())
# Print the chosen number of trading days per year
print('Number of trading days per year :', trading_days_per_year)
```

Finally, we estimate the two parameters μ_{annu} and σ_{annu} by first calculating the cumulative return of the investment over the years. The cumulative return R_{cumu} is defined as

$$R_{cumu} = \frac{P_{ETF}(last\_day) - P_{ETF}(first\_day)}{P_{ETF}(first\_day)} \quad . \tag{2}$$

After calculating the latter, the annualized return can then be found with the formula

$$\mu_{annu} = R_{annu} = (1 + R_{cumu})^{\left(\frac{365}{\Delta_{days}}\right)} - 1 \quad , \tag{3}$$

where 365 is chosen for the number of days in the year, and Δ_{days} is the number of days separating the first and final day of the time series. The annualized volatility can be estimated based on the daily volatility (standard deviation) of the returns σ_{daily} by

$$\sigma_{annu} = \sigma_{daily} \times \sqrt{trading\_days\_per\_year} \quad . \tag{4}$$

Eventually, both properties we are interested in are printed on the command line or inside a log file, depending on the execution of the script. The python block below describes all the operations.

```python
# Show the number of days for the whole period of trading
delta_days = (ETF_data.index.date[-1]-ETF_data.index.date[0]).days
print('Number of days between start and end period :', delta_days)
# Calculate cumulative return of investment
cumu_return = (ETF_data.close[-1] - ETF_data.close[0]) / ETF_data.close[0]
# Estimate annualized return of investment
annual_return = (1 + cumu_return)**(365/delta_days) - 1
# Calculate the average daily return
mean_daily_return = ETF_data['return'].mean()

# Calculate the daily standard deviation (volatility) of returns
daily_sigma = np.std(ETF_data['return'])
# Annualized volatility
annual_sigma = daily_sigma * sqrt(trading_days_per_year)

print()
print()
# Printing both properties we are interested in percentage and rounded numbers
print('Annualized return :', str(round(annual_return,5) * 100) + '%')
print('Annualized volatility :', str(round(annual_sigma,4) * 100) + '%')
```

By the evaluation of the historical data, finally we find:

- $\mu_{annu} = 0.08316$ or μ_{annu} = **8.316%**.
- $\sigma_{annu} = 0.01863$ or σ_{annu} = **18.63%**.

2. Numerical Assessment

In the following section, and for clarity we consider μ_{annu} and σ_{annu} as μ and σ, respectively. Indeed, we select the years as time scale for our simulations, but it can be changed to days or else. To simulate a buy-and-hold investment and predict its value 10 years from now, a Monte Carlo approach coupled with geometric Brownian motion (GBM) is used. The GBM is implemented following different methods for comparison, and verification of the models. Considering a stochastic process S_t, such as the price of a stock (or ETF here), the latter is considered to follow a GBM if it satisfies the following stochastic differential equation (SDE)

$$dS_t = \mu S_t dt + \sigma S_t dW_t = \mu S_t dt + \sigma S_t \xi_t \sqrt{dt} \qquad , \tag{5}$$

where dt is the infinitesimal change of time, $W_t = \xi_t\sqrt{dt}$ is a Wiener process with ξ_t, independent and identically distributed random variables of mean 0 and variance 1. In this study, we consider that the ξ_t follow a normal or Gaussian distribution $\mathcal{N}(0,1)$.

Based on this equation, a model can be implemented (following script) by considering timesteps of size Δt and looping over all timesteps to simulate the GBM from t = 0 to T (final time of the simulation), such as

$$S_{t+\Delta t} = S_t + S_t \times (\mu\Delta t + \sigma\xi_t\sqrt{\Delta t}) \qquad . \tag{6}$$

```python
# This function implements the Monte Carlo methods from the stochastic differential
↪   equation of GBM
def Monte_Carlo_SDE_GBM(S0, sigma, mu, T, dt, n_ETF):
    n_steps = round(T/dt)
    S_price = np.zeros(n_steps) # create an array of n_steps elements
    S_price[0] = n_ETF * S0 # number of ETFs held times price S0 for
    ↪  initialization
    drift = np.zeros(n_steps)
    shock = np.zeros(n_steps)
    for i in range(1, n_steps): # loop for all timesteps
        drift [i] = mu * dt
        shock[i] = sigma * np.random.normal(0,1) * np.sqrt(dt)
        S_price[i] = S_price[i-1] + S_price[i-1] * (drift[i] + shock[i])
    return S_price
```

For an arbitrary initial value S_0, and under Itô's lemma the above SDE has the analytic or **closed form** solution

$$S_t = S_0 \exp\left(\underbrace{\left(\mu - \frac{\sigma^2}{2}\right)t}_{\text{drift}} + \underbrace{\sigma W_t}_{\text{shock}}\right) \quad , \tag{7}$$

where the Wiener process W_t follows a normal distribution with mean = 0 and standard deviation $\sqrt{t}$. The analytic solution is then implemented to return the price of S_T at time T:

```
# This function returns a list values from the direct computation of the analytic
↪   solution at a particular time
def Analytic_Solution_GBM(S0, sigma, mu, T, dt,n_ETF):
    S_prices = list()
    Wt = np.random.normal(loc=0,scale=np.sqrt(T)) # Wt follows a normal
    ↪   distribution with mean = 0, and std = \sqrt(period)
    S_price = n_ETF * S0 * exp((mu - sigma**2 / 2) * T + sigma * Wt )
    S_prices.append(S_price)
    return S_prices
```

Similarly at what has been done for the solving of the SDE with GBM, the GBM can still be captured in the closed form solution by creating a vector of solutions at different timesteps from t = 0 to t = T, and computing the analytic expression at each timestep, and summing W_t to follow the path of Brownian motion. This way, the analytic solution path can be retraced.

```
# This function consists of the computation of the analytical solution with
↪   timesteps to represent the GBM
def Alternative_Monte_Carlo_GBM(S0, sigma, mu, T, dt, n_ETF):
    n_steps = round(T/dt)
    t = np.linspace(0, T, n_steps)
    Wt = np.random.standard_normal(size = n_steps)
    Wt = np.cumsum(Wt)*np.sqrt(dt) # standard brownian motion
    X = (mu-0.5 * sigma**2)*t + sigma*Wt
    S_price = n_ETF*S0*np.exp(X) # geometric brownian motion
    return S_price
```

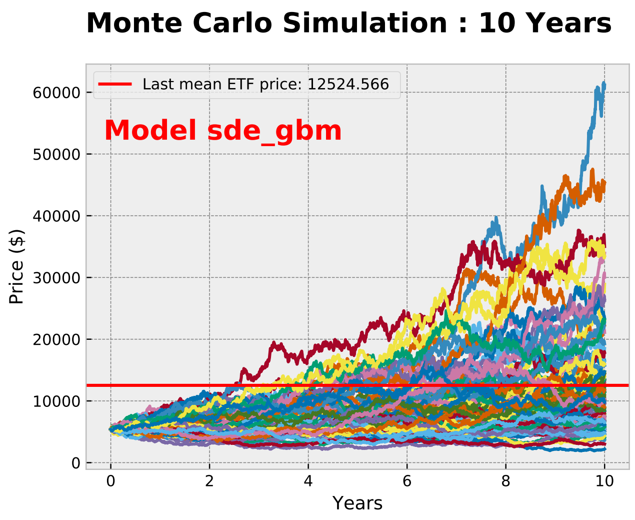
The different algorithms can then be evaluated by running the function below that returns the results of the simulations in a list of matrices. The models need to first be specified in a dictionary, and the number of iterations (for each model) of the Monte Carlo procedure is required. In this study, we take **num_iterations = 30000** which is considered enough based on the mean and standard deviation of final value obtained by the different models after 10 years, and the latter are shown in Table. 1. It can be seen that the average and standard deviation values of the investment after 10 years are close to each other when using the different models, especially for the SDE_GBM and the analytic solution.

	SDE_GBM	Analytic_GBM	Analytic
Mean final value (\$)	22811	23002	22938
Δrel_error_mean (%)	/	0.84	0.56
σ final value (\$)	14606	14880	14661
Δrel_σ (%)	/	1.9	0.38

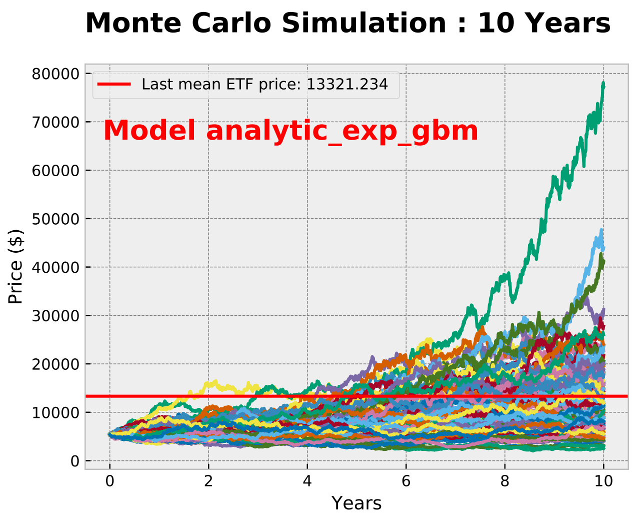
Table 1. Comparison of mean and standard deviation of the final value of investment.

```python
# This function evalutes the different algorithms iteratively and return the
↪  results in a list of dataframes
def evaluate_algorithms(models, num_iterations, S0, sigma, mu, T, dt, n_ETF):
    Matrix_results = list()
    for (name, model) in models.items():
        sim_results = pd.DataFrame()
        for i in range(num_iterations):
            price = model(S0, sigma, mu, T, dt, n_ETF)
            sim_results[i] = price
        Matrix_results.append(sim_results)
    return Matrix_results
```

Fig. 1 shows the evolution of the value of the investment in \$ against the time in years. The function *plot_price_with_time*() from the file “Post_processing.py” creates the plot based on the simulation results for one model. For both models, the randomness of the GBM and discrepancies in the results can be seen alongside the different curves representing the distinct iterations in the Monte Carlo procedure. The final mean value of the ETF investment is also plotted as a horizontal line.



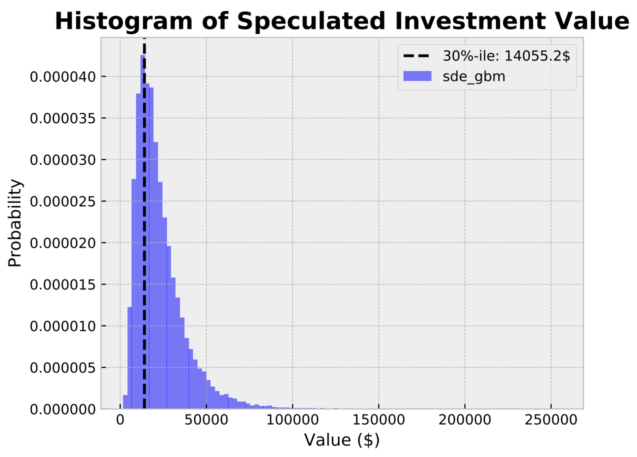
(a) SDE Brownian motion



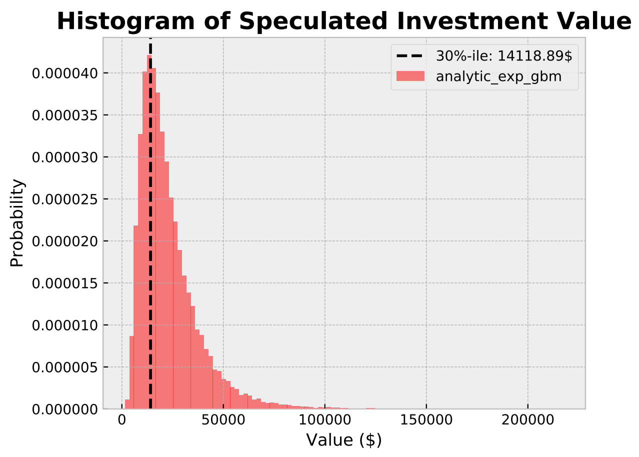
(b) Analytic Brownian motion

Fig. 1. Evolution of the investment value (\$) with time (years) following a Monte Carlo approach.

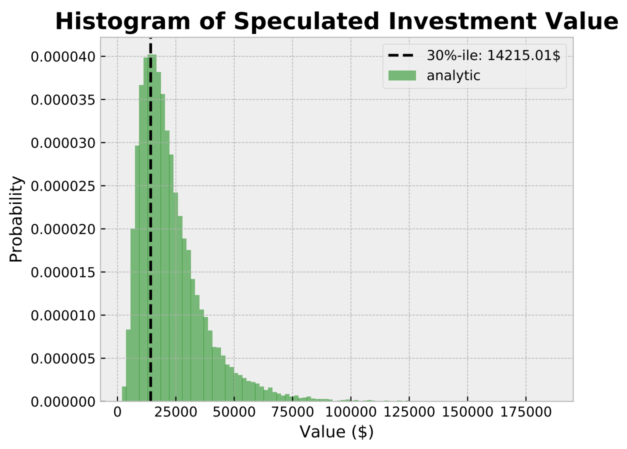
After looking at the evolution with time of the value of the investment, we are interested in evaluating the final value distribution obtained at the end of the 30000 iterations, and also see the discrepancies between the different models. Fig. 2 shows the histogram distributions of the final value of the investment ($) for the different models. It can be seen that the distributions for all models are similar, especially when plotted on the same graph (Fig. 2(d)), and are typical of a log-normal distribution as expected. The 30th percentile is also plotted on the graph as a vertical line as it is an important point for the question 2. The functions *plot_histogram*() and *plot_histogram_together*() from the file "Post_processing.py" generate the different plots.



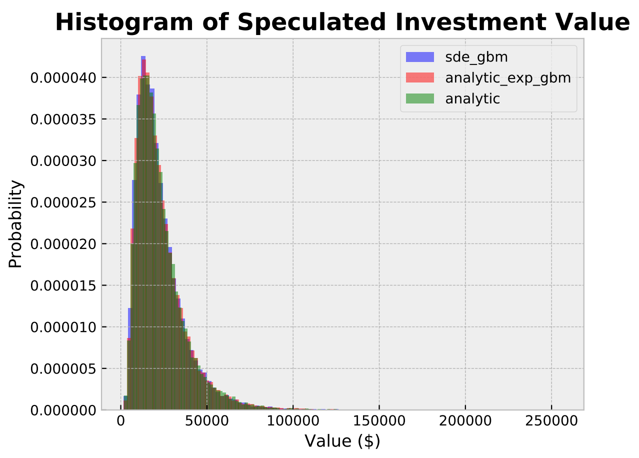
(a) SDE Brownian motion



(b) Analytic Brownian motion



(c) Analytic



(d) All models

Fig. 2. Histogram distributions of investment value ($) after 10 years.

Subsequently, we need to look into percentiles in order to define some probabilities for the investment 10 years from now. The question consists of determining which value the investment can reach with 70% probability. Predicting an exact value is almost impossible considering the randomness and number of simulations. However, with the use of the percentiles we forecast a range for which the investment values lies within, for 70% probability. Table. 2 shows 4 different percentiles that can be used to answer our question. As the results from the different models agree well, we consider the average values $\sim$ 90000 iterations of the numerical procedure. The functions *show_stats*() and *print_percentiles*() return different properties (mean, percentiles, etc) for the final values of investment.

	SDE_GBM	Analytic_GBM	Analytic	Average
15th percentile ($)	10425	10479	10445	10450
30th percentile ($)	14055	14118	14215	14129
70th percentile ($)	26174	26376	26337	26296
85th percentile ($)	35320	35665	35489	35491

Table 2. Comparison of percentiles for the final value of investment.

Denoting a particular investment value (event) V, and its probability $P(V)$, the following statements can be made:

■ $P(V > 30\text{th percentile}) = 0.7$ or $P(V > 14129\$) = 0.7$, meaning that there is 70% chance that the final value of the investment is above 14129$.

■ $P(V \leq 70\text{th percentile}) = 0.7$ or $P(V \leq 26296\$) = 0.7$, meaning that there is 70% chance that the final value of the investment is below or equal to 26296$.

■ $P(V > 15\text{th percentile and } V \leq 85\text{th percentile}) = 0.7$ or $P(V > 10450\$ \text{ and } V \leq 35491\$) = 0.7$, meaning that there is 70% chance that the final value of the investment is the range $]10450\$; 35491\$]$.

The previous results highlight the difficulties when considering the real value of the investment. The judgement criterion is based on the will of the investor: for an averse investor who can freeze its cash flow for 10 years this investment would be worth considering for instance. Nonetheless, based on the simulation results the investment is quite safe with some potential nice return. Actually, for a model improvement the volatility could be considered not constant.

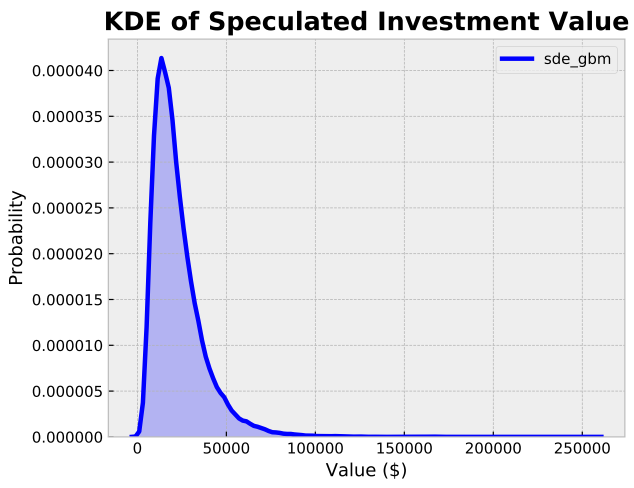
3. Estimation of the probability density function

Finally, the probability density functions (pdf) of the different distributions are assessed using kernel density estimation (kde). The different kdes are plotted using the following script (looped). The seaborn module is used as it is specifically designed for statistical data visualization.

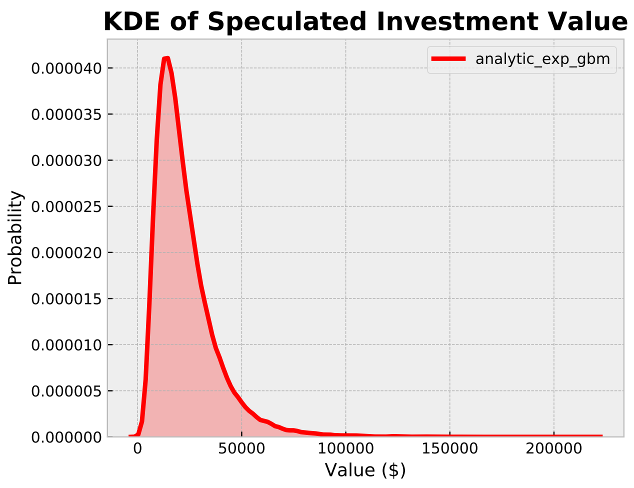
```python
# This function plots the kde of the distribution of final values for the
↪  simulation results
def plot_kde(name, results_simulation, color,save_fig = 1):
    values = results_simulation.values
    final_price = values[-1] # retrieve the final price data --> values of last row
    ↪  of the results dataframe

    sns.distplot(final_price, rug=False, hist =False,
                 kde_kws={"color": str(color), "lw": 3, "shade": True, "label":
                 ↪  str(name)})
                 #hist_kws={"histtype": "step", "linewidth": 3, "alpha": 1, "color":
                 ↪  "g"})

    plt.xlabel('Value (\$)')
    plt.ylabel('Probability')
    plt.title(r'KDE of Speculated Investment Value', fontsize=17,
    ↪  fontweight='bold')
    plt.legend()
```

```python
#Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
if save_fig == 1:
    plt.savefig(str(name) + '_kde.pdf', dpi = 300, bbox_inches='tight')
    plt.close()
else:
    plt.show()
```

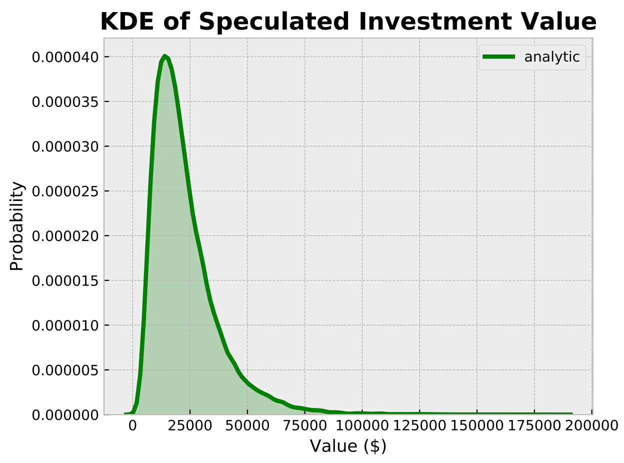
Fig. 3 shows the kde distributions of the final value of the investment ($) for the different models. For each individual kde plot, the area below the curve is also shaded for representing the cumulative distribution function (CDF) with a value of 1. With no surprise, and based from the histogram distributions in Fig. 2, the kde of the different models almost superpose with each other, showing the correlation of the results computed previously.



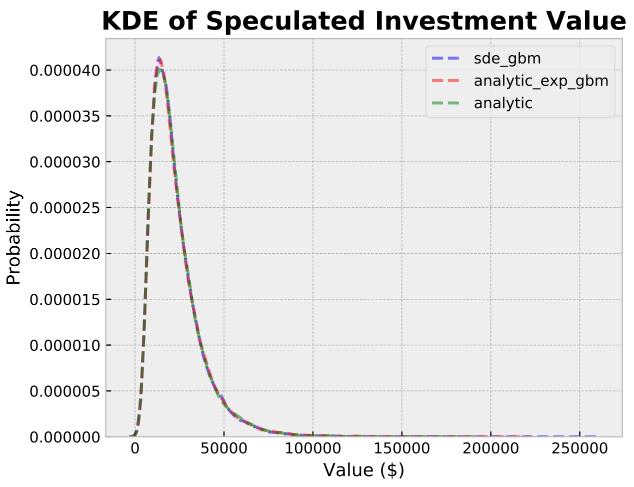
(a) SDE Brownian motion



(b) Analytic Brownian motion



(c) Analytic



(d) 3 models together

Fig. 3. KDE distributions of investment value ($) after 10 years.