```
# Global Tools
import os

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import scipy as sp
import scipy.stats as ss
import yfinance as yf

from cycler import cycler
from datetime import datetime
from ipywidgets import interact, FloatSlider, IntSlider, Dropdown, SelectMultiple, fixed
from IPython.core.interactiveshell import InteractiveShell
from pandas_datareader import DataReader
from scipy.stats import gaussian_kde
from scipy.optimize import minimize
from scipy.signal import find_peaks
from tqdm import tqdm
```

Double-click (or enter) to edit

Global Settings

```
# display all outputs of a cell
# InteractiveShell.ast_node_interactivity = "all"
```

```
# global setting for plt
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['figure.dpi'] = 150
plt.style.use('dark_background')
plt.rcParams['axes.prop_cycle'] = cycler(color=['#76B900', '#DC143C', '#002366', '#5A9BD5', '#A2AAAD', '#B03060'])
```

```
# Data initialization
start_date = datetime(2020, 1, 1)
end_date = datetime(2024, 8, 31)
stock_symbols = ['NVDA', 'AMD', 'INTC', 'QCOM', 'AAPL', 'AVGO']
stocks= yf.download(stock_symbols, start_date, end_date)['Adj Close']
```

```
[*********************100%***********************]  6 of 6 completed
```

Stock Price Curve

```
for ticker in stock_symbols:
    stocks[ticker].plot(label=ticker, linewidth=2)
plt.title("Stock Price History (Jan 2020 - Aug 2024)")
plt.xlabel("Date")
plt.ylabel("Adjusted Close Price")
plt.legend()
plt.grid(True)
plt.show()
```

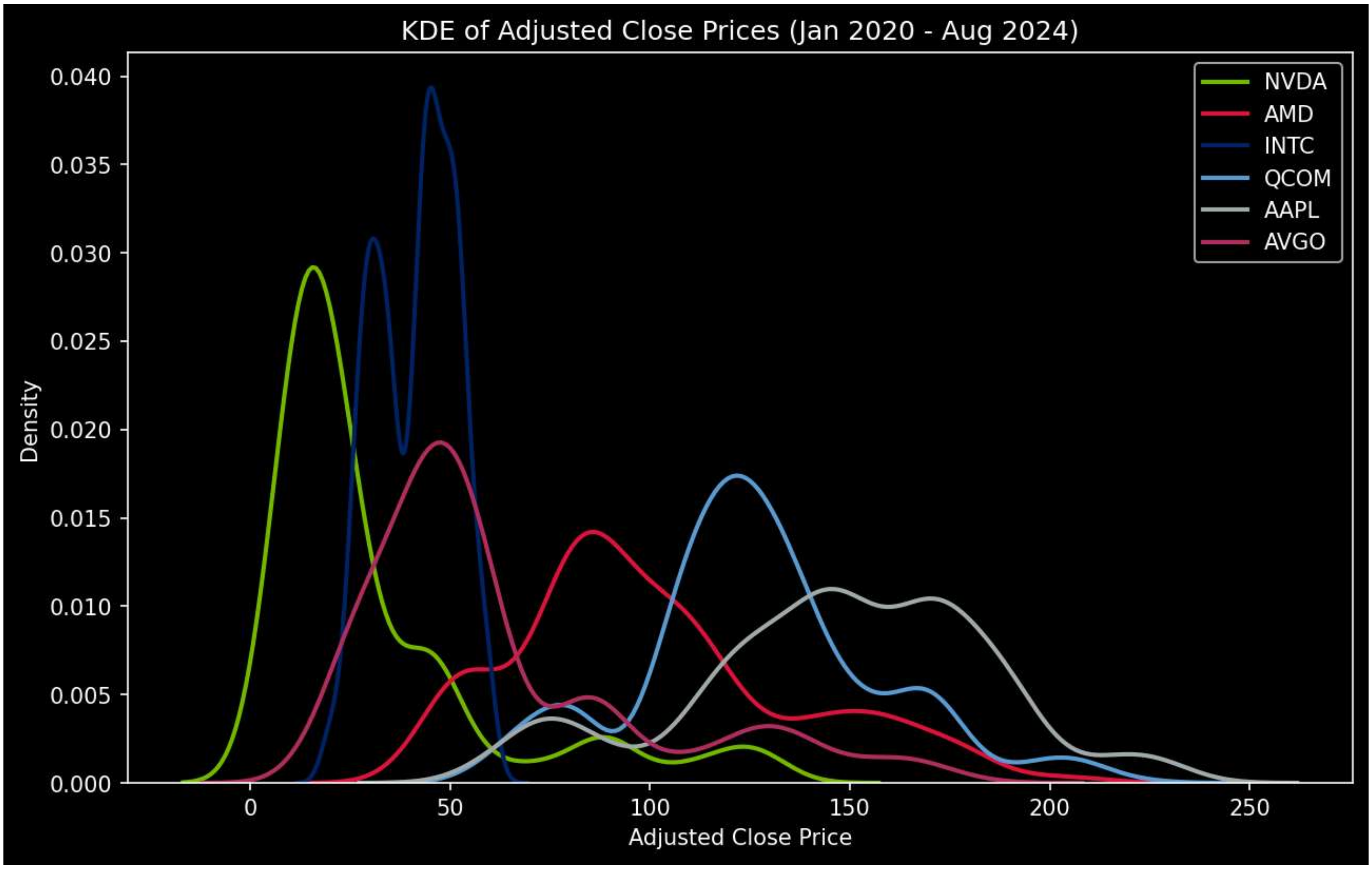
Means, Variance, Skewness, and Kurtosis

```
for ticker in stock_symbols:
    stock_prices = stocks[ticker]
    mean_price = stock_prices.mean()
    variance_price = stock_prices.var()
    skewness_price = ss.skew(stock_prices)
    kurtosis_price = ss.kurtosis(stock_prices)
    print(f"{ticker}:\n Mean: {mean_price}, Variance: {variance_price},\n Skewness: {skewness_price}, Kurtosis: {kurtosis_price}")
```

```
NVDA:
 Mean: 31.72555404749193, Variance: 879.4088272002471,
 Skewness: 1.879951551020386, Kurtosis: 2.7717750794167157
AMD:
 Mean: 100.64815152442841, Variance: 1239.7980894503603,
 Skewness: 0.6734634798536817, Kurtosis: -0.06924435787548067
INTC:
 Mean: 41.4017827588005, Variance: 97.54821990938949,
 Skewness: -0.16579813726860754, Kurtosis: -1.0295164003852713
QCOM:
 Mean: 126.97301707032186, Variance: 936.2664235693105,
 Skewness: 0.30841621175229733, Kurtosis: 0.4168101483271478
AAPL:
 Mean: 146.7570824761204, Variance: 1400.3759206958905,
 Skewness: -0.33844410091045224, Kurtosis: -0.1559720498717856
AVGO:
 Mean: 63.05962385716999, Variance: 1313.863427585109,
 Skewness: 1.3369858506015526, Kurtosis: 0.9558667459887125
```

Kernel Density Estimation (KDE)

```
for ticker in stock_symbols:
    stock_prices = stocks[ticker].dropna()
    sns.kdeplot(stock_prices, label=f'{ticker}', linewidth=2)
plt.title("KDE of Adjusted Close Prices (Jan 2020 - Aug 2024)")
plt.xlabel("Adjusted Close Price")
plt.ylabel("Density")
plt.legend()
plt.show()
```

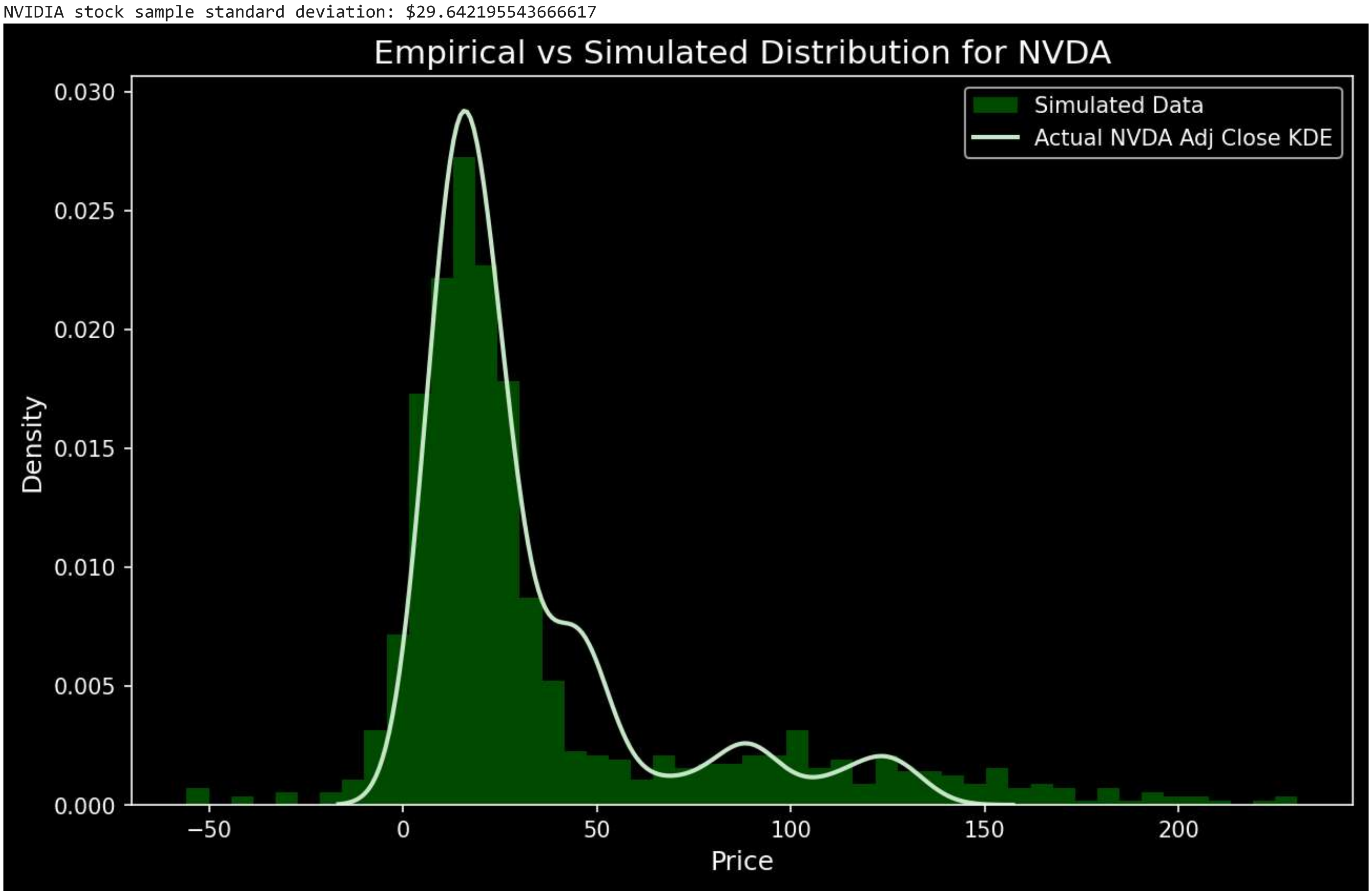


Mixture Modeling

```
X = stocks['NVDA'].values
mu = np.mean(X)
se = np.std(X)
print(f"NVIDIA stock sample mean: ${mu}")
print(f"NVIDIA stock sample standard deviation: ${se}")

# Tuned parameters
mu_1 = 16
sigma1 = 10
mu_2 = 80
sigma2 = 60
p = 0.7
T = 1000

r = np.zeros(T)
for t in range(T):
    eps1 = np.random.normal(0, 1)
    eps2 = np.random.normal(0, 1)
    r1 = mu_1 + sigma1 * eps1
    r2 = mu_2 + sigma2 * eps2
```

```
    u = np.random.uniform(0, 1)
    r[t] = r1 * (u <= p) + r2 * (u > p)
plt.hist(r, bins=50, density=True, alpha=0.6, color='green', label="Simulated Data")
sns.kdeplot(data=X, color='#C8E6C9', label='Actual NVDA Adj Close KDE', lw=2)
plt.title(f'Empirical vs Simulated Distribution for NVDA', fontsize=15)
plt.xlabel('Price', fontsize=12)
plt.ylabel('Density', fontsize=12)
plt.legend()
plt.show()
```

```
NVIDIA stock sample mean: $31.72555404749193
NVIDIA stock sample standard deviation: $29.642195543666617
```



Bootstrap

```
for symbol in stock_symbols:
     X = stocks[symbol].values
     # Bootstrap parameters
     T= X.shape[0]
     B = 1000 # 5000, 100000 [250 9750]
     # using round() instead of int() casting to reduce conversion error
     upper_bound = round(B * 0.975)
     lower_bound = round(B * 0.025)
     mu_boot = np.zeros(B)
     se_boot = np.zeros(B)
     x_boot_std = np.zeros(B)

    # Bootstrap
     for i in range(0, B):
          x_boot = X[np.random.choice(T,T)]
          mu_boot[i] = np.mean(x_boot)
          se_boot[i] = np.std(x_boot)/np.sqrt(T) # std of mu_boot
```

```
        x_boot_std[i] = np.std(x_boot) # std of x_boot
        # CLT: std(x_boost) = sqrt(T)*std(mu_boot)
    mu_boot = np.sort(mu_boot)
    se_boot = np.sort(se_boot)
    xboot_std = np.sort(x_boot_std)

    print(symbol)
    print("(",mu_boot[25],",",
        mu_boot[975],")")
    print("(",se_boot[25],",",
        se_boot[975],")")
    print("(",se_boot[25]*np.sqrt(T),",",
        se_boot[975]*np.sqrt(T),")")
    print("(",xboot_std[25],",",
        xboot_std[975],")")
```

```
NVDA
( 30.094871248865857 , 33.425321846641914 )
( 0.8137015035635372 , 0.9150674913195629 )
( 27.880410698596837 , 31.35358281039724 )
( 27.880410698596837 , 31.35358281039724 )
AMD
( 98.58445480775589 , 102.55167804828494 )
( 0.9839536583125702 , 1.0699144591039187 )
( 33.71387662674903 , 36.65921028970437 )
( 33.71387662674903 , 36.65921028970437 )
INTC
( 40.84268810720622 , 41.98266832263953 )
( 0.27963247601626695 , 0.2970240154325454 )
( 9.581238626026806 , 10.177136826390422 )
( 9.581238626026806 , 10.177136826390422 )
QCOM
( 125.15688278484183 , 128.70924100599663 )
( 0.8533413053499989 , 0.932607663748047 )
( 29.23861631696857 , 31.954573725233566 )
( 29.238616316968567 , 31.954573725233562 )
AAPL
( 144.5799509142692 , 148.77950113676678 )
( 1.0503577512699684 , 1.1337450733265777 )
( 35.98912544417425 , 38.84628224653095 )
( 35.98912544417425 , 38.84628224653094 )
AVGO
( 60.913999862930844 , 65.21170574883585 )
( 1.0015076334480861 , 1.1100402819142299 )
( 34.31534047316912 , 38.034068778562066 )
( 34.31534047316912 , 38.034068778562066 )
```

Monte-Carlo

```
def monte_carlo_simulation(mu_1, sigma1,
                           mu_2, sigma2,
                           p, T=1000):
    r = np.zeros(T)
    for t in range(T):
        eps1 = np.random.normal(0,1,1)
        eps2 = np.random.normal(0,1,1)
        r1 = mu_1 + sigma1 * eps1
        r2 = mu_2 + sigma2 * eps2
        u = np.random.uniform(0,1,1)
        r[t] = r1*(u <= p)+r2*(u > p)

    return r

def simulation_image(parameters, save_path):
    current_X, mu_1, sigma1, mu_2, sigma2, p =  parameters[0], parameters[1], parameters[2],parameters[3], parameters[4], parameters[5]

    plt.figure(figsize=(10,6))
    fig, ax = plt.subplots()
    sns.kdeplot(data=current_X, linewidth=4)
    initalR = monte_carlo_simulation(mu_1, sigma1, mu_2, sigma2, p, T=1000)
    plt.hist(initalR, bins=100, density=True, alpha=0.6, color='green', label="Histogram")
    ax.legend([f'Empirical Kernal Distribution: {save_path}', 'Mixture Model Simulated Distribution'])
    plt.title(f'Monte Carlo Simulation\n,sigma1={sigma1},sigma2={sigma2}')
    ## Save Image
    imagePath = os.path.join(save_path,f"{save_path},sigma1={sigma1},sigma2={sigma2}'.png")

    if not os.path.exists(save_path):
        os.makedirs(save_path)
```

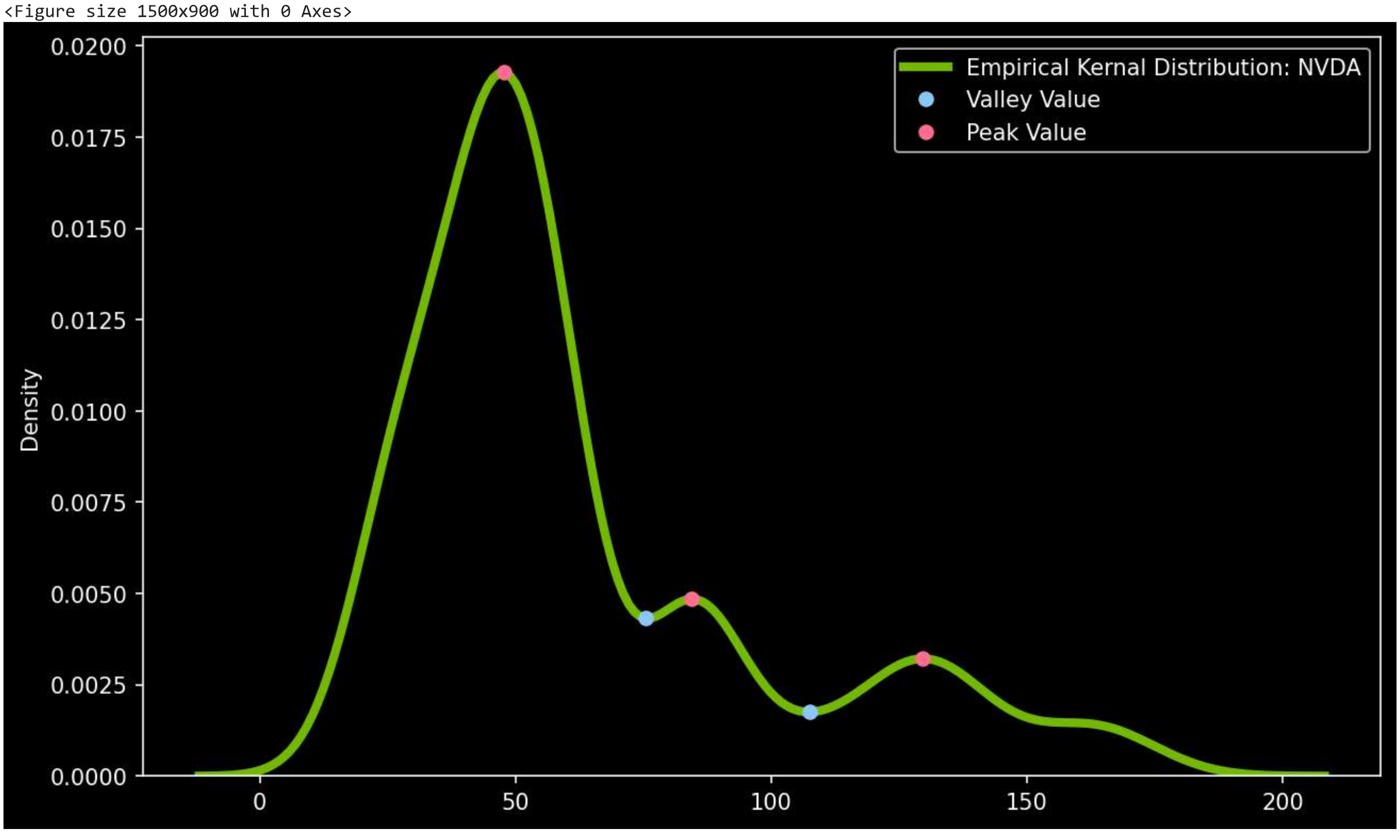
```
    plt.savefig(imagePath)

def calculateOverlap(parameters):

    current_X, mu_1, sigma1, mu_2, sigma2, p =  parameters[0], parameters[1], parameters[2],parameters[3], parameters[4], parameters[5]
    r = monte_carlo_simulation(mu_1, sigma1,
                               mu_2, sigma2,
                          p)

    histArea, bin_edges = np.histogram(r, bins=100, density=True)
    bin_centers = 0.5 * (bin_edges[1:] + bin_edges[:-1])


    kde = gaussian_kde(current_X)
    kdeArea = kde(bin_centers)

    overlap = np.minimum(histArea, kdeArea)
    return scipy.integrate.simps(overlap, bin_centers)

# Loop

def optimize_parameters(parameters, save_path, step=5):
    bestArea = 0
    bestParam = []
    sigma1, sigma2 =  parameters[2], parameters[4]

    if sigma2>sigma1:
        sigma1_range = np.arange(sigma1-step, sigma1+step, 1)
        sigma2_range = np.arange(int(np.floor(sigma2-step*2)), sigma2+step*2, 1)

    else:
        sigma2_range = np.arange(sigma1-step, sigma1+step, 1)
        sigma1_range = np.arange(int(np.floor(sigma2-step*2)), sigma2+step*2, 1)

    for i in tqdm(sigma1_range):
        for j in sigma2_range:
            current_parameters = parameters.copy()
            parameters[2] = i
            parameters[4] = j
            #simulation_image(parameters, save_path)

            overlapArea = calculateOverlap(current_parameters)

            if bestArea < overlapArea:
                bestArea = overlapArea
                bestParam = current_parameters
    return bestParam
```

Get mu_1, mu_2

```
close_data = {}
for symbol in stock_symbols:
    close_data[symbol] = X
current_X = close_data['NVDA']
stock_symbol =  'NVDA'

# Plotting the histogram of r
plt.figure(figsize=(10,6))
fig, ax = plt.subplots()

# Calculate the peak value
sns_kde = sns.kdeplot(data=current_X , linewidth=4)
x, y = sns_kde.get_lines()[0].get_data()

peaks, _ = find_peaks(y)
peak_x_values = x[peaks]

valleys, _ = find_peaks(-y)  # Negate y to find valleys
valleys, _ = find_peaks(-y)

print("Peaks corresponding x values:", x[peaks])
print("Valleys corresponding x values:", x[valleys])

plt.plot(x[valleys], y[valleys], "o", label='Valleys', color = '#90CAF9')
plt.plot(peak_x_values, y[peaks], "ro", label='Peaks', color = '#FF6F91')
```

```
ax.legend([f'Empirical Kernal Distribution: {stock_symbol}',
           'Valley Value',
           'Peak Value'])
```

```
Peaks corresponding x values: [ 47.85874068  84.34171581 129.66904855]
Valleys corresponding x values: [ 75.4973582  107.55815453]
<ipython-input-12-07e99aa1b6e2>:25: UserWarning: color is redundantly defined by the 'color' keyword argument and the fmt string "ro" (-
  plt.plot(peak_x_values, y[peaks], "ro", label='Peaks', color = '#FF6F91')
<matplotlib.legend.Legend at 0x7f81e4130160>
<Figure size 1500x900 with 0 Axes>
```



Calculate P

```
import scipy
```

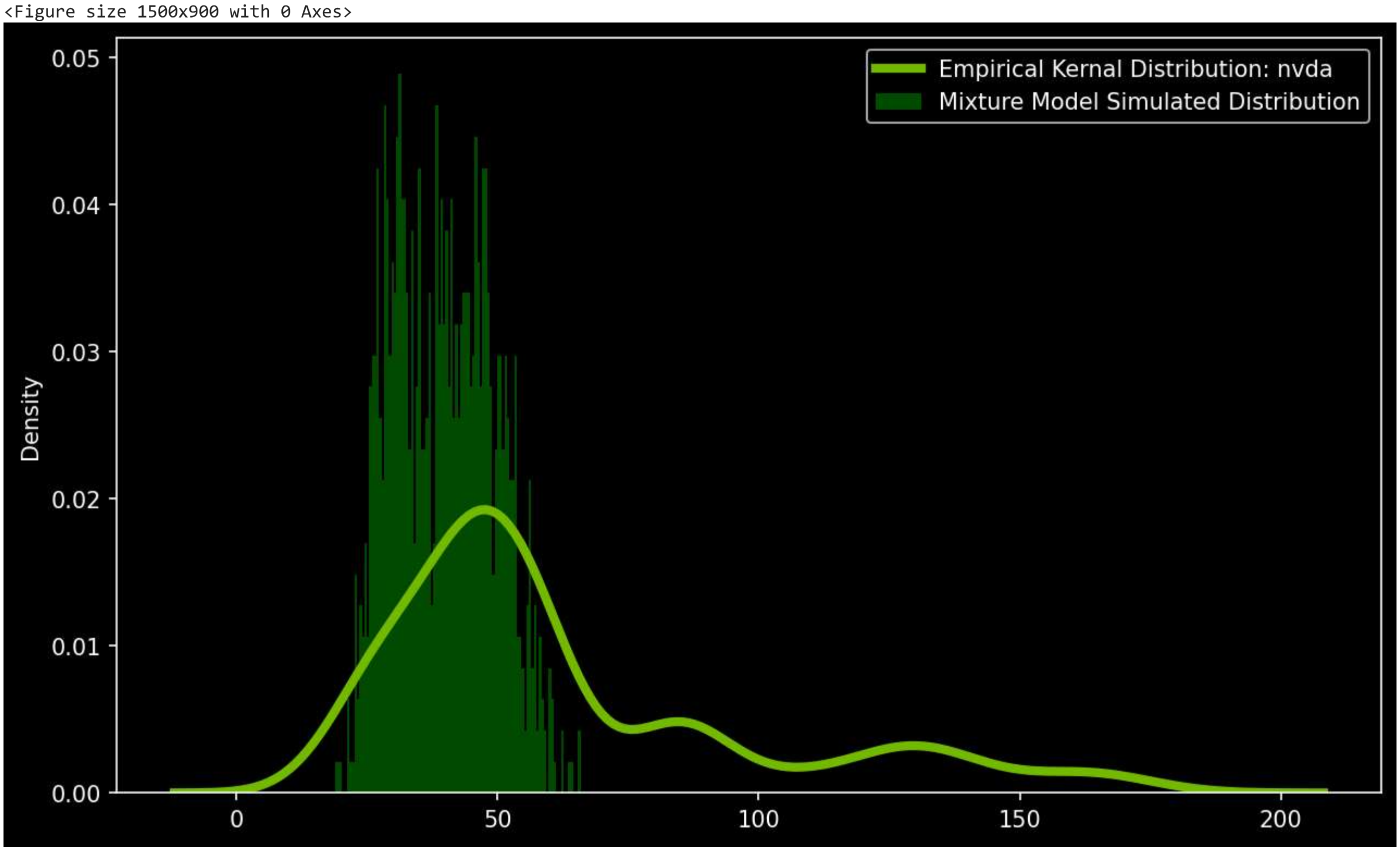
```
cut_point = 68.91934158

# Calculate the total area
total_area = scipy.integrate.simps(y, x)
print("Total Area:", total_area)

# Calculate the specific area between 0 and cut_point
mask = (x >= 0) & (x <= cut_point)
area_between = scipy.integrate.simps(y[mask], x[mask])
print("Specific Area:", area_between)

# Calculate probability
p = area_between / total_area
print("Probability:", p)
```

```
Total Area: 0.999991691115978
Specific Area: 0.7079927965829961
Probability: 0.7079986792619098
```

```
<ipython-input-17-b4e5c7deb908>:4: DeprecationWarning: 'scipy.integrate.simps' is deprecated in favour of 'scipy.integrate.simpson' and
  total_area = scipy.integrate.simps(y, x)
<ipython-input-17-b4e5c7deb908>:9: DeprecationWarning: 'scipy.integrate.simps' is deprecated in favour of 'scipy.integrate.simpson' and
  area_between = scipy.integrate.simps(y[mask], x[mask])
```

```
intcP = [close_data['INTC'], 30, 4, 45, 7, 0.39]
nvdiaP = [close_data['NVDA'], 16, 20, 88, 40, 0.8455528001447512]
parameters = nvdiaP
stock_symbol="nvda"


current_X, mu_1, sigma1, mu_2, sigma2, p =  intcP = close_data['INTC'], 30, 4, 45, 7, 0.35
initalR = monte_carlo_simulation(mu_1, sigma1, mu_2, sigma2, p, T=1000)

plt.figure(figsize=(10,6))
fig, ax = plt.subplots()
sns.kdeplot(data=current_X, linewidth=4)
plt.hist(initalR, bins=100, density=True, alpha=0.6, color='green', label="Histogram")
ax.legend([f'Empirical Kernal Distribution: {stock_symbol}', 'Mixture Model Simulated Distribution'])
```

```
<ipython-input-11-3d6266f22be1>:11: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error i
  r[t] = r1*(u <= p)+r2*(u > p)
<matplotlib.legend.Legend at 0x7f81e3f5d9c0>
<Figure size 1500x900 with 0 Axes>
```



Repeat to tune all parameters

```
nvdiaP = [close_data['NVDA'], 16, 13, 88, 35, 0.8]
amdP = [close_data['AMD'], 86.20013076, 25, 151.50455195, 24, 0.8200047346930525]
intcP = [close_data['INTC'], 30, 4, 45, 7, 0.39]
qcomP = [close_data['QCOM'], 70, 15, 123, 27, 0.15]
aaplP = [close_data['AAPL'], 70, 19, 154, 34,0.15]
```

```
avgoP = [close_data['AVGO'], 45, 19, 128, 34, 0.82]

initialParams = dict(zip(stock_symbols, [nvdiaP, amdP, intcP, qcomP, aaplP, avgoP]))
bestParam = {}
for company, params in initialParams.items():
    best_param = optimize_parameters(params, company, step=7)
    bestParam[company] = best_param
```

```
  0%|          | 0/14 [00:00<?, ?it/s]<ipython-input-11-3d6266f22be1>:11: DeprecationWarning: Conversion of an array with ndim > 0 to a
  r[t] = r1*(u <= p)+r2*(u > p)
<ipython-input-11-3d6266f22be1>:47: DeprecationWarning: 'scipy.integrate.simps' is deprecated in favour of 'scipy.integrate.simpson' and
  return scipy.integrate.simps(overlap, bin_centers)
100%|██████████| 14/14 [00:20<00:00,  1.45s/it]
100%|██████████| 28/28 [00:17<00:00,  1.60it/s]
100%|██████████| 14/14 [00:17<00:00,  1.23s/it]
100%|██████████| 14/14 [00:17<00:00,  1.24s/it]
100%|██████████| 14/14 [00:22<00:00,  1.64s/it]
100%|██████████| 14/14 [00:17<00:00,  1.24s/it]
```

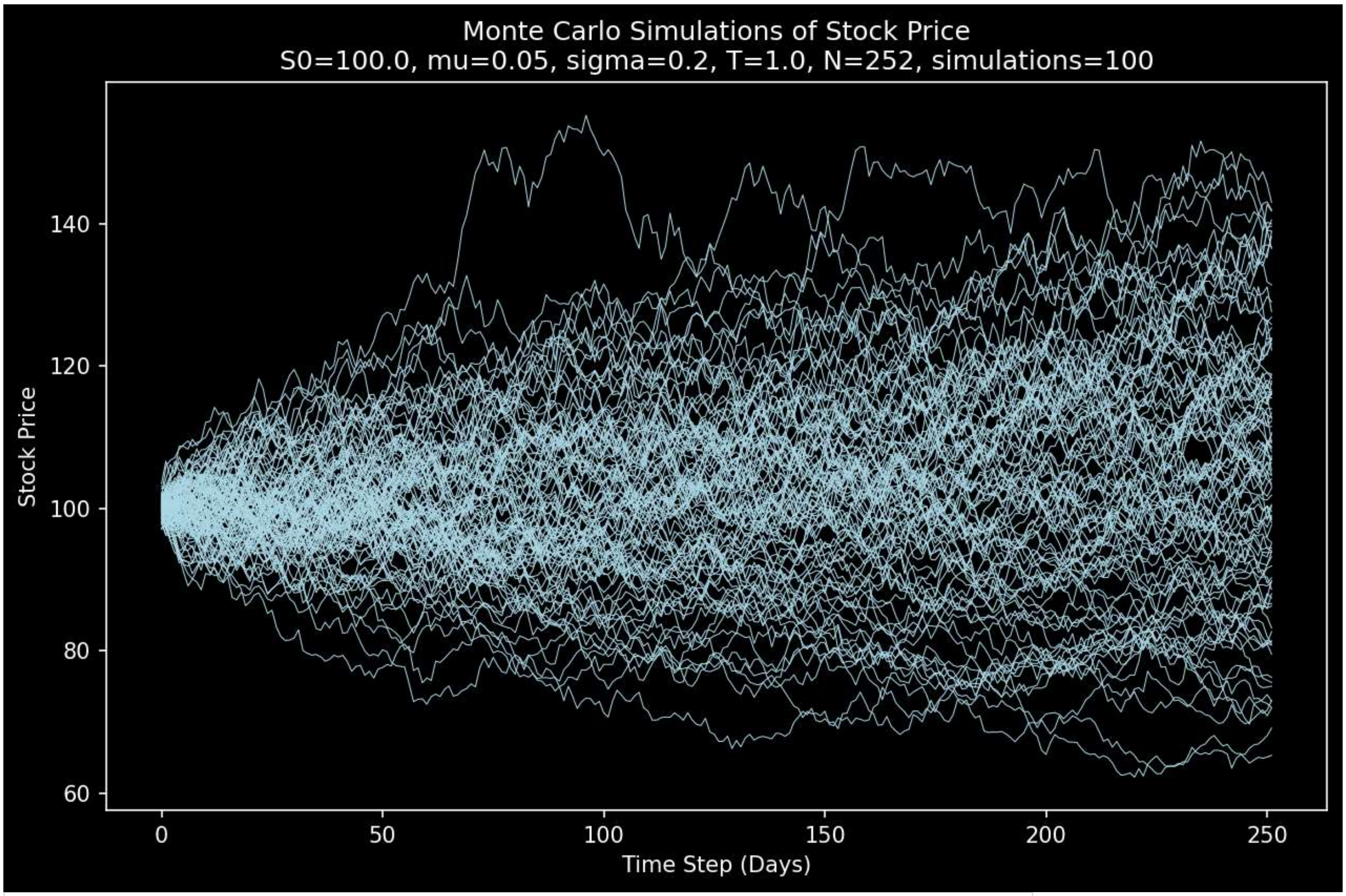
```
# output images are located in "./resources/output.7z"
pd.DataFrame(bestParam).drop(index=0).T
```

	1	2	3	4	5
NVDA	16	19	88	33	0.8
AMD	86.200131	37	151.504552	22	0.820005
INTC	30	10	45	17	0.39
QCOM	70	20	123	39	0.15
AAPL	70	15	154	46	0.15

```
# Define the function for the Monte Carlo stock price simulation
def monte_carlo_stock_price_interactive(S0=100, mu=0.05, sigma=0.2, T=1, N=252, num_simulations=100):
    """
    Interactive Monte Carlo simulation for stock price based on Geometric Brownian Motion.

    Parameters:
    S0: Initial stock price
    mu: Expected return
    sigma: Volatility
    T: Time period (in years)
    N: Number of time steps (daily steps for 1 year)
    num_simulations: Number of simulations

    Returns:
    None (plots the stock price simulations)
    """

    # Time step size
    dt = T / N

    # Running Monte Carlo simulations
    simulations = np.zeros((num_simulations, N))
    for i in range(num_simulations):
        # Generate random changes based on normal distribution
        rand_changes = np.random.normal(mu * dt, sigma * np.sqrt(dt), N)
        # Simulate the price path
        simulations[i, :] = S0 * np.exp(np.cumsum(rand_changes))

    # Plotting the simulations
    plt.figure(figsize=(10, 6))
    for i in range(num_simulations):
        plt.plot(simulations[i, :], color='lightblue', linewidth=0.5)
    plt.xlabel('Time Step (Days)')
    plt.ylabel('Stock Price')
    plt.title(f'Monte Carlo Simulations of Stock Price\nS0={S0}, mu={mu}, sigma={sigma}, T={T}, N={N}, simulations={num_simulations}')
    plt.show()

# Create sliders to change parameters interactively
interact(
    monte_carlo_stock_price_interactive,
```

```
    S0=FloatSlider(value=100, min=50, max=150, step=1, description='Initial Price S0'),
    mu=FloatSlider(value=0.05, min=-0.1, max=0.1, step=0.01, description='Expected Return (mu)'),
    sigma=FloatSlider(value=0.2, min=0.05, max=0.5, step=0.01, description='Volatility (sigma)'),
    T=FloatSlider(value=1, min=0.5, max=5, step=0.5, description='Time Period (T)'),
    N=IntSlider(value=252, min=50, max=500, step=10, description='Time Steps (N)'),
    num_simulations=IntSlider(value=100, min=10, max=500, step=10, description='Simulations')
)
```

Initial Price… 100.00
Expected … 0.05
Volatility (si… 0.20
Time Perio… 1.00
Time Steps… 252
Simulations 100



monte_carlo_stock_price_interactive
def monte_carlo_stock_price_interactive(S0=100, mu=0.05, sigma=0.2, T=1, N=252, num_simulations=100)

```
Interactive Monte Carlo simulation for stock price based on Geometric Brownian Motion.

Parameters:
S0: Initial stock price
mu: Expected return
sigma: Volatility
```