MF796_hw2

February 11, 2024

0.1 MF 796: Computational Methods of Mathematical Finance

Professors Christopher Kelliher and Eugene Sorets Spring 2024

0.1.1 Problem set 2

1. What areas of finance are of most interest to you (e.g. big data, trading, portfolio management, risk management, derivative pricing, analyzing complex derivatives)? I used to be quite interested in futures, options, forex, and interest rates before, because I felt that these financial instruments can reflect the market's views on the global economic landscape relatively truthfully. After more than half a year of studying, I feel that portfolio management and risk management are very intriguing, especially when thinking about using statistical methods, financial knowledge, and algorithms to solve various real-life financial problems, which makes me feel extremely excited. I hope that my job will enable me to keep learning, while also allowing me to apply the knowledge I'm acquiring now.

2. What is the primary reason for your interest in this course? Firstly, because I have an undergraduate background in statistics, I feel quite familiar and comfortable with this kind of fundamental mathematical knowledge and programming tasks. I really hope that I can understand the underlying logic of financial instruments as thoroughly as possible, which requires a solid foundation and accumulation of mathematical skills. This course introduces how various mathematical concepts are applied in financial pricing, and the assignments allow me to experience how these algorithms can be implemented via computers. It is an extremely practical course that deepens my understanding of finance.

3. List all programming languages you have used and your level of familiarity with each.

- **Python**: Proficient
- **R**: Proficient
- **LaTeX**: Intermediate
- **SQL**: Intermediate
- **C**: Intermediate
- **Scala**: Intermediate
- **HTML**: Intermediate

4. Option traders often say that when buying options we get gamma at the expense of theta. What do you think they mean?

- **Gamma** measures the rate of change of an option's delta in relation to the price changes of the underlying asset. Delta itself measures the sensitivity of an option's price to a $1 change in the price of the underlying asset. A high gamma indicates that the delta of an option is highly sensitive to changes in the price of the underlying asset.
- **Theta** measures the rate of decline of an option's price due to the passage of time, holding all else equal. It represents the time decay of an option's value.

When option traders say that "when buying options, we get gamma at the expense of theta," they are highlighting a trade-off between the potential for profit from price movements (gamma) and the guaranteed loss over time due to decay (theta). Buying options provides the buyer with positive gamma, which means they benefit from large price swings in the underlying asset. However, this comes with the cost of theta, or time decay, meaning the value of the option decreases as it gets closer to its expiration date, assuming no other variables change.

In simpler terms, buying options can be advantageous in volatile markets where the price of the underlying asset moves significantly, as the potential gains from those movements (thanks to gamma) can outweigh the costs of time decay (theta). However, if the market does not move as expected, the time decay will erode the option's value, leading to a loss for the option holder. This is the inherent trade-off between gamma and theta when buying options.

5. Evaluation of a known integral using various quadratures: In this problem we are going to compute the price of a European call option using various quadrature methods and compare their properties.
Let our European call option have 3 month expiry, strike 13, and implied vol 25%.
Assume the underlying is 10 now and the interest rate is 5%.

(a) Use Black-Scholes formula to compute the price of the call analytically.

```
[2]: from scipy.stats import norm
     import numpy as np

     # Parameters
     S0 = 10         # Current price of the underlying
     K = 13          # Strike price
     T = 3 / 12      # Time to maturity in years (3 months)
     r = 0.05        # Risk-free interest rate
     sigma = 0.25    # Volatility

     # Black-Scholes formula components
     d1 = (np.log(S0 / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
     d2 = d1 - sigma * np.sqrt(T)

     # Price of the call option
     C = S0 * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)

     C
```

```
[2]: 0.012042247185543142
```

(b) Calculate the price of the call numerically using the following 3 quadrature methods applied to the normal cdf inside the Black-Sholes formula:

- i. Left Riemann rule
- ii. Midpoint rule
- iii. Gauss nodes of your choice (say explicitly why you made that choice) with the number of nodes N = 5, 10, 20, 50 and compute the calculation error as a function of N for each of the methods

[7]:
```
from scipy.special import roots_hermite
from numpy.polynomial.hermite import hermgauss
import scipy.integrate as integrate

def normal_cdf(x):
    """Compute the standard normal CDF."""
    return (1.0 + np.erf(x / np.sqrt(2.0))) / 2.0

def left_riemann_rule(f, a, b, N):
    """Approximate the integral of f from a to b using the Left Riemann Rule␣
 ↪with N subintervals."""
    h = (b - a) / N
    return sum(f(a + i*h) for i in range(N)) * h

def midpoint_rule(f, a, b, N):
    """Approximate the integral of f from a to b using the Midpoint Rule with N␣
 ↪subintervals."""
    h = (b - a) / N
    return sum(f(a + (i + 0.5)*h) for i in range(N)) * h

def gauss_hermite_quad(f, N):
    """Approximate the integral of f using Gauss-Hermite quadrature with N␣
 ↪points."""
    nodes, weights = roots_hermite(N)
    return np.sum(weights * f(nodes * np.sqrt(2)))

# Parameters for numerical integration
a = -10  # Lower bound for integration (approximation for -infinity)
b = 10   # Upper bound for integration (approximation for infinity)
Ns = [5, 10, 20, 50]  # Number of nodes

# Standard normal PDF for numerical integration
std_normal_pdf = lambda x: np.exp(-x**2 / 2) / np.sqrt(2 * np.pi)

# Analytic N(d1) and N(d2)
N_d1_analytic = norm.cdf(d1)
N_d2_analytic = norm.cdf(d2)
```

[8]:
```python
import pandas as pd
# calculate errors for demonstration
errors = {
    'Left Riemann': [],
    'Midpoint': [],
    'Gauss-Hermite': []
}

for N in Ns:
    # Left Riemann Rule
    N_d1_left = left_riemann_rule(std_normal_pdf, a, d1, N)
    N_d2_left = left_riemann_rule(std_normal_pdf, a, d2, N)
    C_left = S0 * N_d1_left - K * np.exp(-r * T) * N_d2_left
    errors['Left Riemann'].append(abs(C_left - C))

    # Midpoint Rule
    N_d1_mid = midpoint_rule(std_normal_pdf, a, d1, N)
    N_d2_mid = midpoint_rule(std_normal_pdf, a, d2, N)
    C_mid = S0 * N_d1_mid - K * np.exp(-r * T) * N_d2_mid
    errors['Midpoint'].append(abs(C_mid - C))

    # Gauss-Hermite Quadrature
    N_d1_gh = gauss_hermite_quad(f, N) * np.exp(-d1**2 / 2)
    N_d2_gh = gauss_hermite_quad(f, N) * np.exp(-d2**2 / 2)
    C_gh = S0 * N_d1_gh - K * np.exp(-r * T) * N_d2_gh
    errors['Gauss-Hermite'].append(abs(C_gh - C))

# Create a DataFrame from the errors dictionary
errors_df = pd.DataFrame(errors, index=Ns)
errors_df.index.name = 'N'

# Display the DataFrame to show errors for different methods across various N␣
 ↪values
errors_df
```

[8]:
```
    Left Riemann  Midpoint  Gauss-Hermite
N
5       0.010643  0.000383       0.012042
10      0.005513  0.000711       0.012042
20      0.002401  0.000236       0.012042
50      0.000846  0.000040       0.012042
```

(c) Estimate the experimental rate of convergence (i.e., as a function of N) of each method and compare it with the known theoretical estimate.

[9]:
```python
# Calculate the experimental rate of convergence
convergence_rates = {}
```

```
for method in errors.keys():
    rates = []
    for i in range(1, len(Ns)):
        N1, N2 = Ns[i-1], Ns[i]
        e_N1, e_N2 = errors[method][i-1], errors[method][i]
        rate = np.log(e_N1 / e_N2) / np.log(N2 / N1)
        rates.append(rate)
    convergence_rates[method] = rates

# Create a DataFrame to display the convergence rates
convergence_df = pd.DataFrame(convergence_rates, index=[f'{Ns[i-1]} to {Ns[i]}'␣
 ↪for i in range(1, len(Ns))])
convergence_df.index.name = 'N Range'

convergence_df
```

[9]:

```
          Left Riemann  Midpoint  Gauss-Hermite
N Range
5 to 10       0.948998 -0.893461   0.000000e+00
10 to 20      1.199331  1.591301  -4.255752e-13
20 to 50      1.138338  1.928094   3.220564e-13
```

(d) Which method is your favorite and why?

0.1.2 Preferred Method: Midpoint Rule

- **Accuracy and Efficiency**: The Midpoint Rule demonstrated significantly better accuracy for a given number of nodes compared to the Left Riemann Rule, as indicated by its lower errors and higher experimental rates of convergence, especially for larger (N). This improvement in accuracy without a substantial increase in computational effort makes it highly efficient for this application.
- **Convergence Rate**: The experimental rates of convergence for the Midpoint Rule approached or exceeded 2 for larger values of (N), suggesting quadratic convergence under the conditions of our problem. This is particularly advantageous when dealing with smooth functions, as it implies that doubling the number of intervals (and thus the computational effort) can potentially reduce the error by a factor of four.
- **Simplicity and Applicability**: While the Gauss-Hermite Quadrature is theoretically more sophisticated and designed for high accuracy in specific cases, its application requires more complex preparation. The Midpoint Rule, on the other hand, is straightforward to implement and does not require such adjustments, making it broadly applicable to a wide range of problems.

[ ]:

```
```