03_european_call_option_pricing

June 24, 2024

1 Pricing European Call Options

1.0.1 Introduction

Suppose a European call option with strike price K and an underlying asset whose spot price at maturity S_T follows a given random distribution. The corresponding payoff function is defined as:

$$\max\{S_T - K, 0\}$$

In the following, a quantum algorithm based on amplitude estimation is used to estimate the expected payoff, i.e., the fair price before discounting, for the option:

$$\mathbb{E}\left[\max\{S_T - K, 0\}\right]$$

as well as the corresponding Δ, i.e., the derivative of the option price with respect to the spot price, defined as:

$$\Delta = \mathbb{P}\left[S_T \geq K\right]$$

The approximation of the objective function and a general introduction to option pricing and risk analysis on quantum computers are given in the following papers:

- Quantum Risk Analysis. Woerner, Egger. 2018.
- Option Pricing using Quantum Computers. Stamatopoulos et al. 2019.

[1]:
```
import matplotlib.pyplot as plt

%matplotlib inline
import numpy as np

from qiskit import QuantumCircuit
from qiskit_algorithms import IterativeAmplitudeEstimation, EstimationProblem
from qiskit.circuit.library import LinearAmplitudeFunction
from qiskit_aer.primitives import Sampler
from qiskit_finance.circuit.library import LogNormalDistribution
```

1.0.2 Uncertainty Model

We construct a circuit to load a log-normal random distribution into a quantum state. The distribution is truncated to a given interval [low, high] and discretized using 2^n grid points, where n denotes the number of qubits used. The unitary operator corresponding to the circuit implements the following:

$$|0\rangle_n \mapsto |\psi\rangle_n = \sum_{i=0}^{2^n-1} \sqrt{p_i}|i\rangle_n,$$

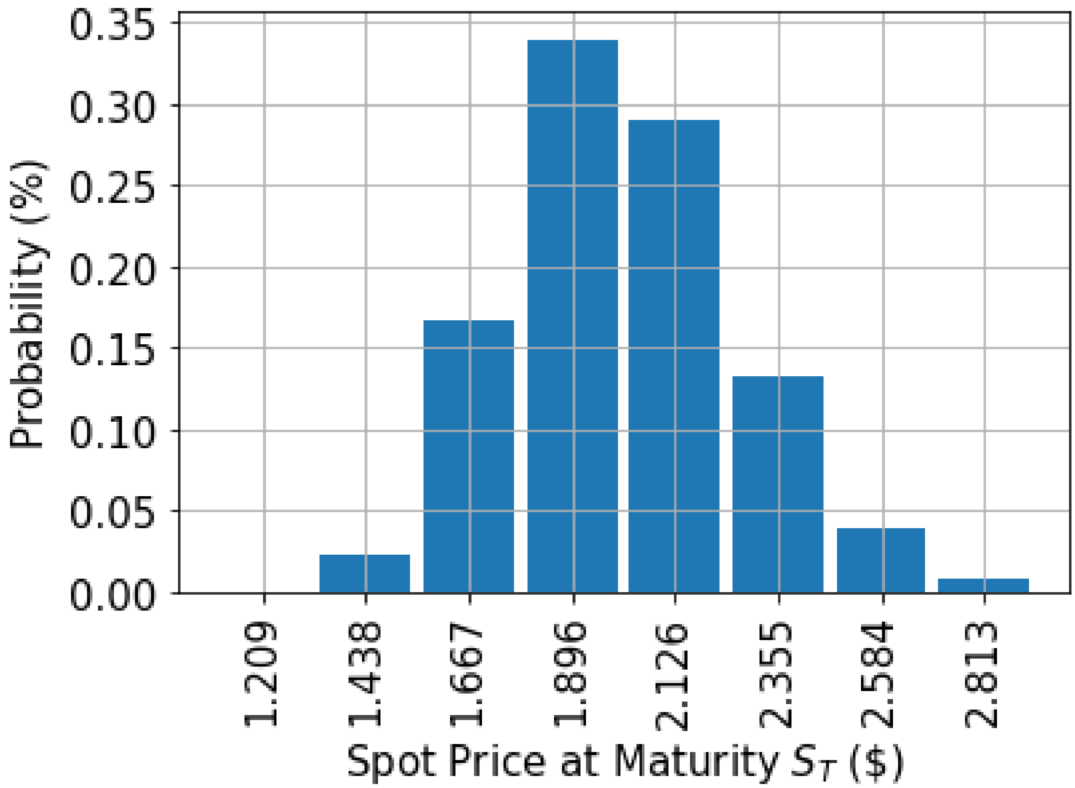
where p_i denote the probabilities corresponding to the truncated and discretized distribution and where i is mapped to the right interval using the affine map:

$$\{0, \dots, 2^n - 1\} \ni i \mapsto \frac{\text{high} - \text{low}}{2^n - 1} * i + \text{low} \in [\text{low}, \text{high}].$$

[2]:
```
# number of qubits to represent the uncertainty
num_uncertainty_qubits = 3

# parameters for considered random distribution
S = 2.0  # initial spot price
vol = 0.4  # volatility of 40%
r = 0.05  # annual interest rate of 4%
T = 40 / 365  # 40 days to maturity

# resulting parameters for log-normal distribution
mu = (r - 0.5 * vol**2) * T + np.log(S)
sigma = vol * np.sqrt(T)
mean = np.exp(mu + sigma**2 / 2)
variance = (np.exp(sigma**2) - 1) * np.exp(2 * mu + sigma**2)
stddev = np.sqrt(variance)

# lowest and highest value considered for the spot price; in between, an␣
↪equidistant discretization is considered.
low = np.maximum(0, mean - 3 * stddev)
high = mean + 3 * stddev

# construct A operator for QAE for the payoff function by
# composing the uncertainty model and the objective
uncertainty_model = LogNormalDistribution(
    num_uncertainty_qubits, mu=mu, sigma=sigma**2, bounds=(low, high)
)
```

[3]:
```
# plot probability distribution
x = uncertainty_model.values
y = uncertainty_model.probabilities
plt.bar(x, y, width=0.2)
```

```
plt.xticks(x, size=15, rotation=90)
plt.yticks(size=15)
plt.grid()
plt.xlabel("Spot Price at Maturity $S_T$ (\$)", size=15)
plt.ylabel("Probability ($\%$)", size=15)
plt.show()
```



1.0.3 Payoff Function

The payoff function equals zero as long as the spot price at maturity S_T is less than the strike price K and then increases linearly. The implementation uses a comparator, that flips an ancilla qubit from $|0\rangle$ to $|1\rangle$ if $S_T \geq K$, and this ancilla is used to control the linear part of the payoff function.

The linear part itself is then approximated as follows. We exploit the fact that $\sin^2(y+\pi/4) \approx y+1/2$ for small $|y|$. Thus, for a given approximation rescaling factor $c_{\text{approx}} \in [0,1]$ and $x \in [0,1]$ we consider

$$\sin^2(\pi/2 * c_{\text{approx}} * (x-1/2) + \pi/4) \approx \pi/2 * c_{\text{approx}} * (x-1/2) + 1/2$$

for small c_{approx}.

We can easily construct an operator that acts as

$$|x\rangle|0\rangle \mapsto |x\rangle \left(\cos(a*x+b)|0\rangle + \sin(a*x+b)|1\rangle\right),$$

using controlled Y-rotations.

Eventually, we are interested in the probability of measuring $|1\rangle$ in the last qubit, which corresponds to $\sin^2(a*x+b)$. Together with the approximation above, this allows to approximate the values of interest. The smaller we choose c_{approx}, the better the approximation. However, since we are then estimating a property scaled by c_{approx}, the number of evaluation qubits m needs to be adjusted accordingly.

For more details on the approximation, we refer to: Quantum Risk Analysis. Woerner, Egger. 2018.

[4]:
```python
# set the strike price (should be within the low and the high value of the␣
↪uncertainty)
strike_price = 1.896

# set the approximation scaling for the payoff function
c_approx = 0.25

# setup piecewise linear objective fcuntion
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(
    num_uncertainty_qubits,
    slopes,
    offsets,
    domain=(low, high),
    image=(f_min, f_max),
    breakpoints=breakpoints,
    rescaling_factor=c_approx,
)

# construct A operator for QAE for the payoff function by
# composing the uncertainty model and the objective
num_qubits = european_call_objective.num_qubits
european_call = QuantumCircuit(num_qubits)
european_call.append(uncertainty_model, range(num_uncertainty_qubits))
european_call.append(european_call_objective, range(num_qubits))

# draw the circuit
european_call.draw()
```

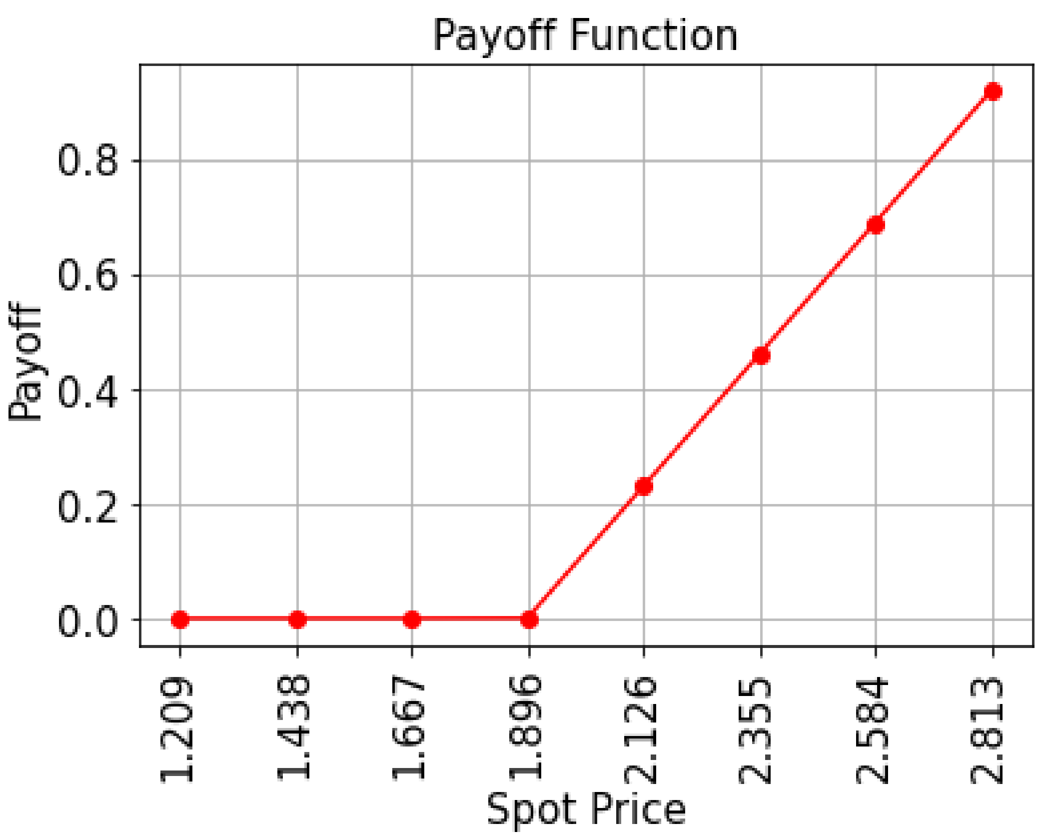
[4]:

```
q_0: 0       0

q_1: 1 P(X)  1
```

```
q_2:  2       2

q_3:       3 F

q_4:       4

q_5:       5

q_6:       6
```

[5]:

```
# plot exact payoff function (evaluated on the grid of the uncertainty model)
x = uncertainty_model.values
y = np.maximum(0, x - strike_price)
plt.plot(x, y, "ro-")
plt.grid()
plt.title("Payoff Function", size=15)
plt.xlabel("Spot Price", size=15)
plt.ylabel("Payoff", size=15)
plt.xticks(x, size=15, rotation=90)
plt.yticks(size=15)
plt.show()
```



[6]:
```
# evaluate exact expected value (normalized to the [0, 1] interval)
exact_value = np.dot(uncertainty_model.probabilities, y)
exact_delta = sum(uncertainty_model.probabilities[x >= strike_price])
print("exact expected value:\t%.4f" % exact_value)
print("exact delta value:   \t%.4f" % exact_delta)
```

```
exact expected value:   0.1623
exact delta value:      0.8098
```

1.0.4 Evaluate Expected Payoff

[7]:
```
european_call.draw()
```

[7]:
```
q_0:  0       0

q_1:  1 P(X)  1

q_2:  2       2

q_3:      3 F

q_4:      4

q_5:      5

q_6:      6
```

[8]:
```
# set target precision and confidence level
epsilon = 0.01
alpha = 0.05

problem = EstimationProblem(
    state_preparation=european_call,
    objective_qubits=[3],
    post_processing=european_call_objective.post_processing,
)
# construct amplitude estimation
ae = IterativeAmplitudeEstimation(
    epsilon_target=epsilon, alpha=alpha, sampler=Sampler(run_options={"shots":␣
 ↪100, "seed": 75})
)
```

[9]:
```
result = ae.estimate(problem)
```

[10]:
```python
conf_int = np.array(result.confidence_interval_processed)
print("Exact value:        \t%.4f" % exact_value)
print("Estimated value:    \t%.4f" % (result.estimation_processed))
print("Confidence interval:\t[%.4f, %.4f]" % tuple(conf_int))
```

```
Exact value:            0.1623
Estimated value:        0.1687
Confidence interval:    [0.1637, 0.1737]
```

Instead of constructing these circuits manually, the Qiskit Finance module offers the `EuropeanCallPricing` circuit, which already implements this functionality as a building block.

[11]:
```python
from qiskit_finance.applications.estimation import EuropeanCallPricing

european_call_pricing = EuropeanCallPricing(
    num_state_qubits=num_uncertainty_qubits,
    strike_price=strike_price,
    rescaling_factor=c_approx,
    bounds=(low, high),
    uncertainty_model=uncertainty_model,
)
```

[12]:
```python
# set target precision and confidence level
epsilon = 0.01
alpha = 0.05

problem = european_call_pricing.to_estimation_problem()
# construct amplitude estimation
ae = IterativeAmplitudeEstimation(
    epsilon_target=epsilon, alpha=alpha, sampler=Sampler(run_options={"shots":
 ↪100, "seed": 75})
)
result = ae.estimate(problem)

conf_int = np.array(result.confidence_interval_processed)
print("Exact value:        \t%.4f" % exact_value)
print("Estimated value:    \t%.4f" % (european_call_pricing.interpret(result)))
print("Confidence interval:\t[%.4f, %.4f]" % tuple(conf_int))
```

```
Exact value:            0.1623
Estimated value:        0.1687
Confidence interval:    [0.1637, 0.1737]
```

1.0.5 Evaluate Delta

The Delta is a bit simpler to evaluate than the expected payoff. Similarly to the expected payoff, we use a comparator circuit and an ancilla qubit to identify the cases where $S_T > K$. However, since we are only interested in the probability of this condition being true, we can directly use this ancilla qubit as the objective qubit in amplitude estimation without any further approximation.

[13]:
```python
from qiskit_finance.applications.estimation import EuropeanCallDelta

european_call_delta = EuropeanCallDelta(
    num_state_qubits=num_uncertainty_qubits,
    strike_price=strike_price,
    bounds=(low, high),
    uncertainty_model=uncertainty_model,
)
```

[14]:
```python
european_call_delta._objective.decompose().draw()
```

[14]:
```
state_0: 0

state_1: 1

state_2: 2
            cmp
state_3: 3

 work_0: 4

 work_1: 5
```

[15]:
```python
european_call_delta_circ = QuantumCircuit(european_call_delta._objective.
 ↪num_qubits)
european_call_delta_circ.append(uncertainty_model,␣
 ↪range(num_uncertainty_qubits))
european_call_delta_circ.append(
    european_call_delta._objective, range(european_call_delta._objective.
 ↪num_qubits)
)

european_call_delta_circ.draw()
```

[15]:
```
q_0: 0       0

q_1: 1 P(X)  1

q_2: 2       2
            ECD
q_3:      3

q_4:      4
```

```
q_5:      5
```

[16]:
```
# set target precision and confidence level
epsilon = 0.01
alpha = 0.05

problem = european_call_delta.to_estimation_problem()

# construct amplitude estimation
ae_delta = IterativeAmplitudeEstimation(
    epsilon_target=epsilon, alpha=alpha, sampler=Sampler(run_options={"shots":␣
 ↪100, "seed": 75})
)
```

[17]:
```
result_delta = ae_delta.estimate(problem)
```

[18]:
```
conf_int = np.array(result_delta.confidence_interval_processed)
print("Exact delta:    \t%.4f" % exact_delta)
print("Estimated value: \t%.4f" % european_call_delta.interpret(result_delta))
print("Confidence interval: \t[%.4f, %.4f]" % tuple(conf_int))
```

```
Exact delta:            0.8098
Estimated value:        0.8091
Confidence interval:    [0.8034, 0.8148]
```

[19]:
```
import tutorial_magics

%qiskit_version_table
%qiskit_copyright
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```