

浙江大学



《量子计算理论基础与软件系统》 实验报告

实验名称 : 量子金融

姓 名 : 王晓宇

学 号 : 3220104364

电子邮箱 : 3220104364@zju.edu.cn

联系电话 : 19550222634

授课教师 : 卢丽强/尹建伟

助 教 : 储天尧

2024 年 12 月 13 日

量子金融

1 问题分析

- 1.1 传统欧式看涨期权定价
- 1.2 量子计算在欧式看涨期权定价中的应用
- 1.3 课程任务

2 算法原理

- 2.1 Black-Scholes 模型
 - 2.1.1 对数正态分布模型
 - 2.1.2 金融数据说明
 - 2.1.3 对数正态分布的参数计算
- 2.2 Black-Scholes模型(更换为正态概率模型)
 - 2.2.1 正态分布模型
 - 2.2.2 金融数据说明
 - 2.2.3 正态分布模型的参数计算
- 2.3 欧式看涨期权定价量子电路实现
 - 2.3.1 欧式看涨期权的一些重要定义
 - 2.3.2 量子算法的目标
 - 2.3.3 实现流程
 - 2.3.3.1 估计期望收益 \bar{E}
 - 2.3.3.2 估计Delta Δ

3 技术实现

- 3.1 导入头文件
- 3.2 搭建对数正态分布模型
- 3.3 绘制不确定性模型的概率分布图
- 3.4 搭建期望收益量子电路
- 3.5 绘制收益与预期现货价格的关系图*
- 3.6 输出精确期望值和精确Delta值*
- 3.7 测试量子电路得到期望收益
- 3.8 返回期望收益测试结果

3.9 导入Qiskit-finance中的Delta比较器实现

3.10 构建Delta量子电路

3.11 测试量子电路得到Delta

3.12 返回Delta测试结果

4 实验测试

4.1 测试不同概率模型

4.1.1 对数正态概率分布

4.1.2 正态概率分布

4.1.3 结果分析

4.2 测试相同概率模型不同参数下的定价结果

4.2.1 模型数据来源的量子比特数

4.2.2 预期收益和Delta计算的精度 ϵ

4.2.3 预期收益和Delta计算的置信度 α

4.2.4 标的资产的当前价格 S_0

4.2.5 波动率 v

4.2.6 无风险利率 r

4.2.7 到期时间 T

5 心得体会

6 选做任务(Bonus)

6.1 算法思路介绍

6.2 数据和表示

6.3 神经网络的定义

6.3.1 量子神经网络 `ansatz` 的定义

6.3.2 量子生成器的定义

6.3.3 经典判别器的定义

6.3.4 创建生成器和判别器

6.4 设置训练循环

6.4.1 损失函数的定义

6.4.2 优化器的定义

6.4.3 训练过程的可视化

6.5 模型训练

6.6 显示训练结果

6.7 欧式看涨期权定价

6.8 测试结果

6.8.1 实际数据

6.8.2 实际训练数据

6.9 结果说明**

7 环境配置

量子金融

1 问题分析

基于 **Python** 的量子金融编程框架初探——期权定价应用

难点：阅读文档、熟悉 [Qiskit](#) 和 [Qiskit Finance](#)

Qiskit Finance是目前做的相对较为成熟的软件包。[example1](#) 中展示了一个投资组合优化问题的简单例子，大致过程是将投资组合优化问题转化为一个二次规划问题，再用QAOA求解。

量子金融（**Quantum Finance**）是量子计算与金融领域的交叉学科，旨在利用量子计算的强大计算能力解决传统金融中的复杂问题，例如期权定价、风险管理、投资组合优化等。欧式看涨期权定价是量子金融中的一个经典问题，传统上使用 **Black-Scholes** 模型 进行计算，而量子计算则提供了新的方法来加速这一过程。

1.1 传统欧式看涨期权定价

在传统金融中，欧式看涨期权的定价通常使用 **Black-Scholes** 模型，其公式为：

$$C = S_0 N(d_1) - X e^{-rT} N(d_2)$$

其中：

- C : 看涨期权的价格。
- S_0 : 标的资产的当前价格。
- X : 行权价格。
- r : 无风险利率。
- T : 到期时间。
- σ : 波动率。
- $N(d)$: 标准正态分布的累积分布函数。

尽管 **Black-Scholes** 模型在理论上非常有效，但在实际应用中存在以下问题：

1. 计算复杂性：

- 对于高维问题（如多资产期权），计算复杂度会显著增加。

- 需要精确计算标准正态分布的累积分布函数 $N(d)$ 。
2. 波动率估计：
- 波动率 σ 通常是根据历史数据估计的，可能存在误差。
3. 市场假设：
- Black-Scholes 模型假设市场无摩擦、波动率恒定等，这些假设在现实中不一定成立。
-

1.2 量子计算在欧式看涨期权定价中的应用

量子计算通过其强大的并行计算能力和量子算法，可以加速欧式看涨期权的定价过程。

量子 Black-Scholes 模型

量子计算可以加速 Black-Scholes 模型中的关键计算步骤，例如：

- 标准正态分布的累积分布函数 $N(d)$ ：
 - 传统计算需要数值积分或查表，而量子算法（如量子 Monte Carlo 方法）可以更快地估计 $N(d)$ 。

量子 Monte Carlo 方法

Monte Carlo 方法是一种通过随机采样估计期望值的方法，广泛用于金融衍生品的定价。量子 Monte Carlo 方法利用量子叠加态的并行性，可以显著加速模拟过程。

量子近似优化算法 (QAOA)

QAOA 是一种量子算法，可以用于优化问题。在期权定价中，QAOA 可以用于：

- 优化波动率的估计。
- 解决多资产期权定价中的组合优化问题。

1.3 课程任务

1. 仿照 [example1](#)，用 [Qiskit Finance](#) 做一个欧式看涨期权定价的简单流程：
将原问题转化为一个振幅估计问题，再用振幅估计的电路求解；

第二、三章节解决
2. 测试不同的概率模型以及相同概率模型不同参数下的定价结果；

第四章节第一、二小节解决

- 清楚理解算法中各变量的含义，探究算法中参数对最终结果的影响，定性地或定量地。

第四章节第二小节解决

2 算法原理

2.1 Black-Scholes 模型

在欧式看涨期权的定价中，对数正态分布模型（Log-Normal Model）是基于 Black-Scholes 模型的核心假设之一。

Black-Scholes 模型假设标的资产（如股票）的价格服从对数正态分布。

2.1.1 对数正态分布模型

对数正态分布是一种连续概率分布，其随机变量的对数服从正态分布。在金融中，标的资产的价格 S_t 通常被假设为对数正态分布，即：

$$\ln(S_t) \sim \mathcal{N}(\mu, \sigma^2)$$

其中：

- $\ln(S_t)$: 标的资产价格的对数。
- $\mathcal{N}(\mu, \sigma^2)$: 正态分布，均值为 μ ，方差为 σ^2 。

2.1.2 金融数据说明

在 Black-Scholes 模型中，对数正态分布的参数主要包括：

1. 标的资产的当前价格 S_0
2. 行权价格 X : 期权在到期日可以执行的价格。
3. 无风险利率 r : 无风险投资的年化收益率（通常使用国债收益率）。
4. 波动率 v : 标的资产价格的年化波动率，表示价格变化的波动程度。
5. 到期时间 T : 期权的剩余到期时间，以年为单位。

2.1.3 对数正态分布的参数计算

先给结论：

μ : 对数正态分布的均值参数。

$$\mu = \ln(S_0) + \left(r - \frac{v^2}{2} \right) T$$

σ : 对数正态分布的标准差参数。

$$\sigma = v\sqrt{T}$$

对数正态分布的均值（期望值）即 S_t 的均值估计：

$$E(S_t) = e^{\mu + \frac{\sigma^2}{2}}$$

对数正态分布的方差即 S_t 的方差：

$$\text{Var}(S_t) = e^{2\mu + \sigma^2} (e^{\sigma^2} - 1)$$

其标准差：

$$\sqrt{\text{Var}(S_t)} = \sqrt{e^{2\mu + \sigma^2} (e^{\sigma^2} - 1)}$$

在对数正态分布模型中，变量 S_t 服从对数正态分布，即 S_t 的对数服从正态分布：

$$\ln(S_t) \sim \mathcal{N}(\mu, \sigma^2)$$

$$\mu = \ln(S_0) + \left(r - \frac{v^2}{2} \right) T$$

$$\sigma = v\sqrt{T}$$

其中：

- μ : 对数正态分布的均值参数。
- σ^2 : 对数正态分布的方差参数。

对数正态分布的均值（期望值）即 S_t 的均值估计：

$$E(S_t) = e^{\mu + \frac{\sigma^2}{2}}$$

推导过程：

1. 对数正态分布的密度函数为：

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln(x)-\mu)^2}{2\sigma^2}}, \quad x > 0$$

2. 均值的计算公式为：

$$E(X) = \int_0^\infty x f(x) dx$$

3. 通过变量替换和积分计算，可以得到：

$$E(X) = e^{\mu + \frac{\sigma^2}{2}}$$

对数正态分布的方差即 S_t 的方差：

$$\text{Var}(S_t) = e^{2\mu+\sigma^2} (e^{\sigma^2} - 1)$$

其标准差：

$$\sqrt{\text{Var}(S_t)} = \sqrt{e^{2\mu+\sigma^2} (e^{\sigma^2} - 1)}$$

推导过程：

1. 方差的计算公式为：

$$\text{Var}(X) = E(X^2) - [E(X)]^2$$

2. 计算 $E(X^2)$ ：

$$E(X^2) = e^{2\mu+2\sigma^2}$$

3. 代入均值公式：

$$\text{Var}(X) = e^{2\mu+2\sigma^2} - \left(e^{\mu+\frac{\sigma^2}{2}}\right)^2 = e^{2\mu+\sigma^2} (e^{\sigma^2} - 1)$$

3σ 原则 (Three-Sigma Rule)

3σ 原则 (Three-Sigma Rule) 是统计学中的一个重要概念，用于描述正态分布（或对数正态分布）的特性。根据 3σ 原则，对于一个正态分布 $\ln(S_t) \sim \mathcal{N}(\mu, \sigma^2)$ ，随机变量 S_t 的取值范围大约在 $\mu - 3\sigma$ 到 $\mu + 3\sigma$ 之间，覆盖了约 99.7% 的数据

根据 3σ 原则，对数正态分布的最小值为：

因为现货价格不能为负，确保最低值不会低于0，这里使用了max函数

$$S_{t\min} = \max \left(0, E(S_t) - 3\sqrt{\text{Var}(S_t)} \right)$$

根据 3σ 原则，对数正态分布的最大值为：

$$S_{t \max} = E(S_t) + 3\sqrt{\text{Var}(S_t)}$$

- 通过计算对数正态分布的参数，可以进一步使用 **Black-Scholes** 公式计算欧式看涨期权的价格。

这些参数的计算和理解对于期权定价和风险管理至关重要。

2.2 Black-Scholes模型(更换为正态概率模型)

在欧式看涨期权的定价中，如果将对数正态模型替换为正态模型（即假设标的资产的价格 S_t 服从正态分布），则需要重新计算相关参数。

2.2.1 正态分布模型

在正态分布模型中，标的资产的价格 S_t 被假设为服从正态分布：

$$S_t \sim \mathcal{N}(\mu_S, \sigma_S^2)$$

其中：

- μ_S : 标的资产价格的均值。
- σ_S : 标的资产价格的标准差。

2.2.2 金融数据说明

在 **Black-Scholes** 模型中，对数正态分布的参数主要包括：

- 标的资产的当前价格 S_0
- 行权价格 X : 期权在到期日可以执行的价格。
- 无风险利率 r : 无风险投资的年化收益率（通常使用国债收益率）。
- 波动率 v : 标的资产价格的年化波动率，表示价格变化的波动程度。
- 到期时间 T : 期权的剩余到期时间，以年为单位。

2.2.3 正态分布模型的参数计算

标的资产价格的均值 μ_S :

在正态分布模型中，标的资产价格的均值 μ_S 可以通过以下公式计算：

$$\mu_S = S_0 e^{rT}$$

正态分布模型假设标的资产的价格在到期日的期望值为 $S_0 e^{rT}$ ，即标的资产的当前价格按无风险利率增长。

标的资产价格的标准差 σ_S

在正态分布模型中，标的资产价格的标准差 σ_S 可以通过以下公式计算：

$$\sigma_S = v\sqrt{T}$$

正态分布模型假设标的资产价格的标准差与时间 T 的平方根成正比。

2.3 欧式看涨期权定价量子电路实现

我们现在来讨论将欧式看涨期权定价也即Black-Scholes 模型用量子电路实现的主体思路

2.3.1 欧式看涨期权的一些重要定义

欧式看涨期权是一种金融衍生品，赋予持有者在到期日 T 以行权价格 K 购买标的资产的权利。

- 收益函数：

$$\text{Payoff}(S_T) = \max\{S_T - K, 0\}$$

其中：

- S_T : 标的资产在到期日的价格。
- K : 行权价格。

- 期望收益-- \mathbb{E}

- 期望收益是期权在到期日的平均收益，即：

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

其中 \mathbb{E} 表示期望值。

期望收益是期权定价的核心，表示期权的公平价格（未折现）。

- 风险相关参数--**Delta**

- **Delta** 是期权价格对标的资产价格的敏感性，定义为：

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

其中 \mathbb{P} 表示概率。

- Delta 表示标的资产价格 S_T 大于或等于行权价格 K 的概率，是期权定价中的重要参数，用于风险管理与对冲策略。
- Delta 主要用于描述标的资产价格变动时，期权价格的变化量， $\Delta = \text{期权价格变化值} / \text{标的资产价格变化值}$

2.3.2 量子算法的目标

- 估计期望收益 \mathbb{E} ：

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

- 估计**Delta**：

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

2.3.3 实现流程

在估计期望收益 \mathbb{E} 时，我们使用了以下步骤

1. 实现一个酉操作 (Unitary Operator)，将初始量子态 $|0\rangle_n$ 映射到目标量子态 $|\psi\rangle_n$ ——将对数正态分布 (Log-Normal Distribution) 加载到量子态，使得量子态的振幅对应于分布的概率密度 (此时现价截断到一个有限的区间 $[\text{low}, \text{high}]$ ，此时的分布离散化为 2^n 个网格点，其中 n 是量子比特的数量)
2. 引入一位辅助比特，并实现比较器量子电路实现：如果 $S_T \geq K$ ，则将一个辅助量子比特从 $|0\rangle$ 翻转为 $|1\rangle$ 。
3. 使用受控 Y 旋转门，产生变换： $|x\rangle|0\rangle \mapsto |x\rangle(\cos(ax + b)|0\rangle + \sin(ax + b)|1\rangle)$
4. 此时测量辅助比特 $|1\rangle$ 的振幅 $\sin(ax + b)^2$ ，再通过相应逆变换将三角方法中的概率转化成线性式中的概率即可

在估计 Delta Δ 时，我们使用了以下步骤

1. 实现一个酉操作 (Unitary Operator)，将初始量子态 $|0\rangle_n$ 映射到目标量子态 $|\psi\rangle_n$ ——将对数正态分布 (Log-Normal Distribution) 加载到量子态，使得量子态的振幅对应于分布的概率密度 (此时现价截断到一个有限的区间 $[low, high]$ ，此时的分布离散化为 2^n 个网格点，其中 n 是量子比特的数量)
2. 引入一位辅助比特，并实现比较器量子电路实现：如果 $S_T \geq K$ ，则将一个辅助量子比特从 $|0\rangle$ 翻转为 $|1\rangle$ 。
3. 此时直接测量辅助比特 $|1\rangle$ 的振幅即可

2.3.3.1 估计期望收益 E

1. 对数正态分布模型的加载并实现截断和离散化

利用 `LogNormalDistribution` 方法构建对数正态分布模型 (本质构建的是一个酉操作，将初始量子态 $|0\rangle_n$ 映射到目标量子态 $|\psi\rangle_n$)，指定参数时，将截断范围等等参数传入函数，使得量子态的振幅对应于分布的概率密度

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

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

2. 引入辅助比特并构建比较器

利用 `LinearAmplitudeFunction` 类的实现，我们得到一个专属于对期望收益测量的电路——如果 $S_T \geq K$ ，则将一个辅助量子比特从 $|0\rangle$ 翻转为 $|1\rangle$ 。

在量子计算中，直接实现线性函数 $S_T - K$ 是困难的。

参考论文：[Quantum risk analysis | npj Quantum Information \(nature.com\)](https://www.nature.com/articles/s41534-020-0044-1)

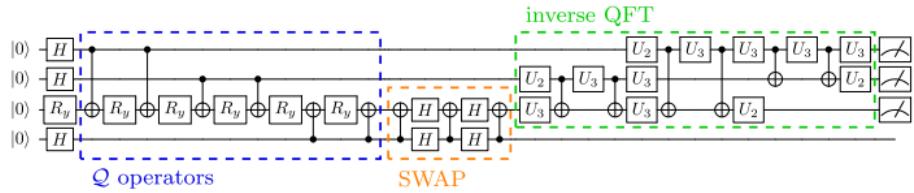
利用三角函数的近似性质，将线性函数近似为：

$$\sin^2 \left(\frac{\pi}{2} c_{\text{approx}} \left(x - \frac{1}{2} \right) + \frac{\pi}{4} \right) \approx \frac{\pi}{2} c_{\text{approx}} \left(x - \frac{1}{2} \right) + \frac{1}{2}$$

其中：

- c_{approx} ：近似缩放因子，取值范围为 $[0, 1]$ 。
- x ：标准化后的标的资产价格，取值范围为 $[0, 1]$ 。

此时我们通过近似，即将近似过的数据 $|\psi\rangle_n$ 转化为 $|x\rangle$ ，此时使用受控 Y 旋转门来实现如下变换即可得到等价：



$$|x\rangle|0\rangle \mapsto |x\rangle (\cos(ax + b)|0\rangle + \sin(ax + b)|1\rangle)$$

3. 测量辅助比特|1⟩的振幅 $\sin(ax + b)^2$ 。

由于 [LinearAmplitudeFunction](#) 类中已实现相应的逆变换，我们这里仅需要利用迭代振幅估计器 [IterativeAmplitudeEstimation](#) 实现测量即可

2.3.3.2 估计DeltaΔ

1. 对数正态分布模型的加载并实现截断和离散化

利用 LogNormalDistribution 方法构建对数正态分布模型（本质构建的是一个酉操作，将初始量子态 $|0\rangle_n$ 映射到目标量子态 $|\psi\rangle_n$ ），指定参数时，将截断范围等等参数传入函数，使得量子态的振幅对应于分布的概率密度

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

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

2. 引入辅助比特并构建比较器

利用 EuropeanCallDelta 类的实现，我们得到一个专属于对 Delta 测量的电路——如果 $S_T \geq K$ ，则将一个辅助量子比特从 $|0\rangle$ 翻转为 $|1\rangle$ 。

在量子计算中，直接实现线性函数 $S_T - K$ 是困难的。与期望测量方法中提到的思路不同的是，我们这里仅对满足条件的概率感兴趣，因此我们无需通过三角式进行近似等价，只需要利用受控 Y 门实现满足条件后翻转目标比特即可

$$|x\rangle|0\rangle \mapsto |x\rangle \left(\sqrt{1 - \Delta}|0\rangle + \sqrt{\Delta}|1\rangle \right)$$

3. 测量辅助比特的振幅得到Delta

利用迭代振幅估计器 [IterativeAmplitudeEstimation](#) 实现测量即可

3 技术实现

3.1 导入头文件

```
1 import matplotlib.pyplot as plt
2
3 %matplotlib inline
4 import numpy as np
5
6 from qiskit import QuantumCircuit
7 from qiskit_algorithms import IterativeAmplitudeEstimation,
EstimationProblem
8 from qiskit.circuit.library import LinearAmplitudeFunction
9 from qiskit.primitives import Sampler as RefSampler
10 from qiskit_finance.circuit.library import LogNormalDistribution,
UniformDistribution, NormalDistribution,
GaussianConditionalIndependenceModel
```

3.2 搭建对数正态分布模型

为量子振幅估计 (QAE) 构建不确定性模型，用于欧式看涨期权定价。

```
1 # 不确定模型的量子比特数`num_uncertainty_qubits`，这个值越大，精度越高，但是计算量也越大
2 num_uncertainty_qubits = 3
3
4 # 期权的参数(考虑的随机分布的参数)
5 S = 2.0 # 初始化股票价格
6 vol = 0.4 # 波动率
7 r = 0.05 # 无风险利率
8 T = 40 / 365 # 到期时间
9
10 # 由对数正态分布得到的参数
11 mu = (r - 0.5 * vol**2) * T + np.log(S)
12 sigma = vol * np.sqrt(T)
13 mean = np.exp(mu + sigma**2 / 2)
14 variance = (np.exp(sigma**2) - 1) * np.exp(2 * mu + sigma**2)
15 stddev = np.sqrt(variance)
```

```

16 # 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
17 low = np.maximum(0, mean - 3 * stddev)
18 high = mean + 3 * stddev
19
20
21 # 由正态分布得到的参数
22 mean_normal = s * np.exp(r * T)
23 stddev_normal = vol * np.sqrt(T)
24 # 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
25 low_normal = np.maximum(0, mean_normal - 3 * stddev_normal)
26 high_normal = mean_normal + 3 * stddev_normal
27
28 # 构建QAE的A算子，通过组合不确定模型和目标函数
29 # 这里给出若干种不确定模型的构建方式(主要是正态分布和对数正态分布)
30 uncertainty_model = LogNormalDistribution(
31     num_uncertainty_qubits, mu=mu, sigma=sigma**2, bounds=(low,
32     high)
33 )
34 # uncertainty_model = NormalDistribution(
35 #     num_uncertainty_qubits, mu=mean_normal,
36 #     sigma=stddev_normal**2, bounds=(low_normal, high_normal)
37 # )
38
39 # 待完善
40 # uncertainty_model = UniformDistribution(
41 #     num_uncertainty_qubits
42 # )
43 # n_normal_gauss = 1
44 # uncertainty_model = GaussianConditionalIndependenceModel(
45 #     n_normal_gauss, 1, sigma=sigma**2, bounds=(low, high)
# )

```

这里注释了许多不同的概率分布模型，分别有正态、对数正态、均值、高斯独立分布的概率分布，这里我引入了正态模型以替换Black-Scholes模型中的对数正态概率分布模型，以完成实验要求。

| 有更多概率分布模型的探索不知道会不会更新写进来(逃

首先, 定义了一个变量 `num_uncertainty_qubits`, 表示用于表示不确定性的量子比特数量, 这里设置为 3 个量子比特。

接下来, 定义了一些参数来描述随机分布:

- `S`: 初始现货价格, 设置为 2.0。
- `vol`: 波动率, 设置为 0.4 (即 40%)。
- `r`: 年利率, 设置为 0.05 (即 4%)。
- `T`: 到期时间, 设置为 40 天 (以年为单位, 即 40/365)。

然后, 计算对数正态分布的参数:

- `mu`: 计算公式为 `(r - 0.5 * vol**2) * T + np.log(S)`, 表示对数正态分布的均值。
- `sigma`: 计算公式为 `vol * np.sqrt(T)`, 表示对数正态分布的标准差。
- `mean`: 计算公式为 `np.exp(mu + sigma**2 / 2)`, 表示对数正态分布的均值。
- `variance`: 计算公式为 `(np.exp(sigma**2) - 1) * np.exp(2 * mu + sigma**2)`, 表示对数正态分布的方差。
- `stddev`: 计算公式为 `np.sqrt(variance)`, 表示对数正态分布的标准差。

接下来, 确定现货价格的最低和最高值, 并在此范围内进行等距离散化:

- `low`: 计算公式为 `np.maximum(0, mean - 3 * stddev)`, 表示现货价格的最低值。
- `high`: 计算公式为 `mean + 3 * stddev`, 表示现货价格的最高值。

最后, 构建用于 QAE 的不确定性模型 `uncertainty_model`, 这是通过组合不确定性模型和目标函数来实现的。这里使用 `LogNormalDistribution` 类来表示对数正态分布, 并传入了量子比特数量、均值、方差以及现货价格的范围。

主要目的是为量子振幅估计构建一个不确定性模型, 以便在后续的量子计算中使用该模型进行欧式看涨期权的定价。

3.3 绘制不确定性模型的概率分布图

```
1 # 画出不确定模型的概率分布
2 x = uncertainty_model.values
3 y = uncertainty_model.probabilities
4 plt.bar(x, y, width=0.2)
5 plt.xticks(x, size=15, rotation=90)
6 plt.yticks(size=15)
7 plt.grid()
8 plt.xlabel("Spot Price at Maturity $S_T (\$)", size=15)
9 plt.ylabel("Probability (\%)", size=15)
10 plt.show()
```

这段代码用于绘制不确定性模型的概率分布图，以可视化对数正态分布的结果。

- 首先，`x = uncertainty_model.values` 和 `y = uncertainty_model.probabilities` 分别获取不确定性模型中的值和对应的概率。
 - 这些值表示在期权到期时可能的现货价格，而概率表示每个现货价格出现的可能性。
- 接下来，使用 `plt.bar()` 绘制条形图，其中 `x` 代表横轴上的现货价格，`y` 代表纵轴上的概率，条形的宽度设置为 `0.2`。
- 然后，使用 `plt.xticks()` 设置横轴刻度，其中 `x` 为刻度位置，`size` 设置刻度标签的字体大小为 `15`，`rotation=90` 将刻度标签旋转 `90` 度，以便更好地显示。
- 类似地，使用 `plt.yticks()` 设置纵轴刻度的字体大小为 `15`。
`plt.grid()` 添加网格线，以便更容易读取图表中的值。
- 接下来，使用 `plt.xlabel()` 设置横轴标签，标签内容为“`Spot Price at Maturity $T (\$)`”，并将字体大小设置为 `15`。同样，使用 `plt.ylabel()` 设置纵轴标签，标签内容为“`Probability (%)`”，并将字体大小设置为 `15`。
- 最后，使用 `plt.show()` 显示绘制的图表。

主要目的是通过条形图直观地展示不确定性模型的概率分布，帮助理解在期权到期时不同现货价格的可能性。

3.4 搭建期望收益量子电路

这段代码展示了如何为欧式看涨期权定价设置目标函数，并构建相应的量子电路。

```
1 # 设置期权的执行价格(应该在不确定性模型的最低值和最高值之间)
2 strike_price = 1.896
3
4 # 设置目标函数的近似缩放
5 c_approx = 0.25
6
7 # 设置分段线性目标函数
8 breakpoints = [low, strike_price]
9 slopes = [0, 1]
10 offsets = [0, 0]
11 f_min = 0
12 f_max = high - strike_price
13 european_call_objective = LinearAmplitudeFunction(
14     num_uncertainty_qubits,
15     slopes,
16     offsets,
17     domain=(low, high),
18     image=(f_min, f_max),
19     breakpoints=breakpoints,
20     rescaling_factor=c_approx,
21 )
22
23 # 通过组合不确定模型和目标函数构建QAE的A算子
24 num_qubits = european_call_objective.num_qubits
25 european_call = QuantumCircuit(num_qubits)
26 european_call.append(uncertainty_model,
27     range(num_uncertainty_qubits))
28 european_call.append(european_call_objective, range(num_qubits))
29 european_call.draw()
```

首先，设置了期权的执行价格 (`strike price`)，这里设置为 `1.896`。执行价格应在不确定性范围的最低值和最高值之间。

接下来，设置了用于近似支付函数的缩放因子 `c_approx`，这里设置为 `0.25`。

然后，设置分段线性目标函数的参数：

- `breakpoints`：断点，设置为 `[low, strike_price]`，表示在这些点上函数的斜率会发生变化。
- `slopes`：斜率，设置为 `[0, 1]`，表示在不同区间内的斜率值。
- `offsets`：偏移量，设置为 `[0, 0]`，表示在不同区间内的偏移量。
- `f_min`：函数的最小值，设置为 `0`。
- `f_max`：函数的最大值，设置为 `high - strike_price`。

使用这些参数，创建了一个 `LinearAmplitudeFunction` 实例

`european_call_objective`，它表示欧式看涨期权的目标函数。这个函数将不确定性模型的输出映射到支付函数的值。

接下来，构建用于量子振幅估计（QAE）的 `A` 操作符，通过组合不确定性模型和目标函数来实现。首先，获取目标函数所需的量子比特数量 `num_qubits`。然后，创建一个量子电路 `european_call`，并将不确定性模型和目标函数分别附加到电路中。

最后，使用 `european_call.draw()` 绘制量子电路图，以可视化电路的结构。

主要目的是设置和构建用于欧式看涨期权定价的量子电路，通过量子振幅估计来计算期权的价格。

3.5 绘制收益与预期现货价格的关系图*

```
1 # 绘制期权的精确收益函数(但是这里会由于量子比特数量过少造成离散点的斜率不
  对齐的问题，执行价格附近最为明显)
2 x = uncertainty_model.values
3 y = np.maximum(0, x - strike_price)
4 plt.plot(x, y, "ro-")
5 plt.grid()
6 plt.title("Payoff Function", size=15)
7 plt.xlabel("Spot Price", size=15)
8 plt.ylabel("Payoff", size=15)
9 plt.xticks(x, size=15, rotation=90)
10 plt.yticks(size=15)
11 plt.show()
```

| 绘制欧式看涨期权的精确收益函数，展示期权在不同现货价格下的收益情况。

首先，代码从 `uncertainty_model` 中获取现货价格的值，并将其赋值给变量 `x`。然后，计算每个现货价格下的期权收益，并将结果赋值给变量 `y`。收益计算公式为 `np.maximum(0, x - strike_price)`，即现货价格减去执行价格的差值，如果差值为负则收益为零。

通过这段代码查看欧式看涨期权在不同现货价格下的收益情况。然而，由于量子比特数量较少，可能会导致离散点的斜率不对齐，特别是在执行价格附近最为明显。

3.6 输出精确期望值和精确Delta值*

```
1 # 计算精确的期望值(归一化到[0,1])和Delta值
2 exact_value = np.dot(uncertainty_model.probabilities, y)
3 exact_delta = sum(uncertainty_model.probabilities[x >=
4     strike_price])
5 print("exact expected value:\t%.4f" % exact_value)
6 print("exact delta value: \t%.4f" % exact_delta)
```

| 计算并输出欧式看涨期权的精确期望值和精确 Delta 值。

首先，计算精确期望值 `exact_value`，这是通过对不确定性模型的概率分布和目标函数值进行点积计算得到的。具体来说，`np.dot` 计算了不确定性模型中每个可能现货价格的概率与对应的支付函数值的乘积之和。这个期望值被归一化到 `[0, 1]` 区间。

接下来，计算精确 `Delta` 值 `exact_delta`，这是通过对所有现货价格大于等于执行价格的概率进行求和得到的。具体来说，`sum()` 计算了所有现货价格大于等于执行价格的概率之和。`Delta` 值表示期权价格对现货价格变化的敏感度。

最后，使用 `print` 函数输出计算结果。

主要目的是通过计算和输出精确期望值和 `Delta` 值，帮助理解欧式看涨期权在给定不确定性模型下的定价和风险特性。

3.7 测试量子电路得到期望收益

```
1 # 通过设置epsilon和alpha来控制估计的精度和置信度
2 epsilon = 0.01 # 1%的精度
3 alpha = 0.05 # 95%的置信度
```

```
4 # 我们要测量的量子比特的索引应该是量子比特数(因为python的索引是从0开始
5 # 的)
6
7 problem = EstimationProblem(
8     state_preparation=european_call,
9     objective_qubits=[num_uncertainty_qubits],
10    post_processing=european_call_objective.post_processing,
11 )
12 # 构建振幅估计器
13 ae = IterativeAmplitudeEstimation(
14     epsilon_target=epsilon, alpha=alpha,
15     sampler=RefSampler(options={"shots": 100, "seed": 75})
16 )
17 result = ae.estimate(problem)
```

设置和构建用于欧式看涨期权定价的量子振幅估计算法。

- 首先，设置目标精度 `epsilon` 和置信水平 `alpha`。
 - `epsilon` 被设置为 `0.01`，表示估计目标的精度；
 - `alpha` 被设置为 `0.05`，表示置信水平，即结果的置信区间为 `95%`。
- 接下来，创建一个 `EstimationProblem` 实例 `problem`。这个实例包含了所有运行振幅估计算法所需的特定问题信息：
 - `state_preparation`：量子电路 `european_call`，用于准备输入状态。
 - `objective_qubits`：目标量子比特，这里设置为 `[num_uncertainty_qubits]`，表示测量的量子比特索引。
 - `post_processing`：后处理函数，这里使用 `european_call_objective.post_processing`，用于将算法结果映射到目标区间。
- 然后，构建一个 `IterativeAmplitudeEstimation` 实例 `ae`，用于执行迭代量子振幅估计。这个实例使用以下参数：
 - `epsilon_target`：目标精度，设置为 `epsilon`。
 - `alpha`：置信水平，设置为 `alpha`。

- `sampler`: 采样器，这里使用 `Sampler`类，并设置运行选项 `{"shots": 100, "seed": 75}`，表示每次运行的采样次数为 `100`，随机种子为 `75`。

主要目的是准备和配置量子振幅估计算法，以便在后续步骤中运行并获取期权定价结果。

3. `result = ae.estimate(problem)`调用了 `ae` 对象的 `estimate` 方法，并将 `problem`作为参数传递给该方法。

- `ae` 对象是一个实现了振幅估计算法的实例。

`estimate`方法的作用是对传入的 `problem`（一个 `EstimationProblem` 实例）执行振幅估计算法。该方法会返回一个 `IterativeAmplitudeEstimationResult` 对象，其中包含了估计结果和相关的统计数据。

具体来说，`estimate` 方法会执行以下步骤：

1. 检查是否提供了采样器（sampler），如果没有提供，则使用默认的 Qiskit 采样器。
2. 初始化一些用于算法的变量，例如幂次列表、比率列表、theta 区间和置信区间等。
3. 通过循环迭代，不断调整和计算估计值，直到满足精度要求。
4. 在每次迭代中，构建量子电路并运行测量，计算测量结果的概率，并更新置信区间和 theta 区间。
5. 最终，计算出估计值及其置信区间，并将这些结果存储在 `IterativeAmplitudeEstimationResult` 对象中返回。

通过调用 `ae.estimate(problem)` 的振幅估计结果，并将其存储在 `result` 变量中。

3.8 返回期望收益测试结果

```

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

```

首先将 `result.confidence_interval_processed` 转换为一个 NumPy 数组，并将其赋值给变量 `conf_int`。

`result.confidence_interval_processed` 是一个包含置信区间的对象，通过使用 `np.array` 函数将其转换为 NumPy 数组，可以方便地进行数值计算和操作。

接下来，代码使用 `print` 函数输出三个信息：

1. 精确值 (`exact_value`)
2. 估计值 (`result.estimation_processed`)
3. 置信区间 (`conf_int`)

通过这种方式，代码清晰地输出了精确值、估计值和置信区间，便于用户查看和分析这些结果。

3.9 导入Qiskit-finance中的Delta比较器实现

```
1 from qiskit_finance.applications.estimation import EuropeanCallDelta
2
3 european_call_delta = EuropeanCallDelta(
4     num_state_qubits=num_uncertainty_qubits,
5     strike_price=strike_price,
6     bounds=(low, high),
7     uncertainty_model=uncertainty_model,
8 )
9 european_call_delta._objective.decompose().draw()
```

1. 首先从 `qiskit_finance.applications.estimation` 模块中导入了 `EuropeanCallDelta` 类。
 - `EuropeanCallDelta` 类是一个用于估算欧式看涨期权 `Delta` 值的应用类。`Delta` 是金融领域中的一个重要指标，用于衡量期权价格相对于基础资产价格变化的敏感度。
2. 代码创建了一个 `EuropeanCallDelta` 类的实例，命名为 `european_call_delta`。在实例化过程中，传递了以下参数：
 - `num_state_qubits`：表示用于表示随机变量的量子比特数量，这里使用了变量 `num_uncertainty_qubits`。
 - `strike_price`：欧式期权的执行价格，这里使用了变量 `strike_price`。
 - `bounds`：离散化随机变量的边界元组 `(min, max)`，这里使用了变量 `low` 和 `high`。

- `uncertainty_model`：用于编码问题分布的量子电路，这里使用了变量 `uncertainty_model`。

`EuropeanCallDelta` 类的构造函数会根据这些参数初始化一个欧式看涨期权 `Delta` 问题的实例。创建一个 `EuropeanCallDeltaObjective` 对象来表示目标函数，并将 `uncertainty_model` 和目标函数组合成一个量子电路 `self._state_preparation`。

这个量子电路用于在量子计算机上执行期权 `Delta` 值的估算。

3.10 构建Delta量子电路

```

1 european_call_delta_circ =
2     QuantumCircuit(european_call_delta._objective.num_qubits)
3 european_call_delta_circ.append(uncertainty_model,
4     range(num_uncertainty_qubits))
5 european_call_delta_circ.append(
6     european_call_delta._objective,
7     range(european_call_delta._objective.num_qubits))
8
9 european_call_delta_circ.draw()

```

1. 首先创建了一个 `QuantumCircuit` 实例 `european_call_delta_circ`，其量子比特数量由 `european_call_delta._objective.num_qubits` 决定。
 - `QuantumCircuit` 是 `Qiskit` 中用于构建和操作量子电路的类。
2. 接下来，代码使用 `append` 方法将 `uncertainty_model` 添加到电路中。
 - `uncertainty_model` 是一个量子电路或门，表示不确定性模型。
 - `range()` 指定了该模型应用到的量子比特范围，其中 `num_uncertainty_qubits` 是不确定性模型所需的量子比特数量。
3. 然后，代码再次使用 `append` 方法将 `european_call_delta._objective` 添加到电路中。
 - `european_call_delta._objective` 是一个量子电路或门，表示欧式看涨期权 `Delta` 值的目标函数。

- `range()` 指定了该目标函数应用到的量子比特范围。

最后，调用 `draw` 方法生成电路的可视化表示，便于用户查看和理解电路结构。

3.11 测试量子电路得到Delta

```

1 # 设置目标精度和置信度
2 epsilon = 0.01
3 alpha = 0.05
4
5 problem = european_call_delta.to_estimation_problem()
6
7 # 构建振幅估计器
8 ae_delta = IterativeAmplitudeEstimation(
9     epsilon_target=epsilon, alpha=alpha,
10    sampler=RefSampler(options={"shots": 100, "seed": 75}))
11
12 result_delta = ae_delta.estimate(problem)

```

- 首先设置了目标精度 `epsilon` 和置信水平 `alpha`。
 - `epsilon` 表示估计值的目标精度，这里设置为 `0.01`；`alpha` 表示置信水平，这里设置为 `0.05`。
- 接下来，代码通过调用 `to_estimation_problem()` 方法，将欧式看涨期权 `Delta` 值问题转换为一个 `EstimationProblem` 实例，并将其赋值给变量 `problem`。
 - `to_estimation_problem` 方法会将欧式看涨期权 `Delta` 值问题的量子电路和目标函数封装到 `EstimationProblem` 对象中，以便后续的振幅估计算法使用。
- 然后，代码构建了一个 `IterativeAmplitudeEstimation` 实例 `ae_delta`，用于执行迭代振幅估计算法。
 - 构造函数中传递了目标精度 `epsilon`、置信水平 `alpha` 和一个 `RefSampler` 实例。
 - `RefSampler` 用于在量子电路上运行测量，并设置了运行选项 `shots` 为 `100` 和 `seed` 为 `75`，以确保测量的重复性和随机性。

4. 运行电路调用了 `ae_delta` 对象的 `estimate` 方法，并将 `problem` 作为参数传递给该方法。

- `ae_delta` 是一个 `IterativeAmplitudeEstimation` 类的实例，用于执行迭代振幅估计算法。
- `estimate` 方法的作用是对传入的 `problem` (一个 `EstimationProblem` 实例) 执行振幅估计算法。
- 该方法会返回一个 `IterativeAmplitudeEstimationResult` 对象，其中包含了估计结果和相关的统计数据。

具体来说，`estimate` 方法会执行以下步骤：

1. 检查是否提供了采样器 (sampler)，如果没有提供，则使用默认的 Qiskit 采样器。
2. 初始化一些用于算法的变量，例如幂次列表、比率列表、theta 区间和置信区间等。
3. 通过循环迭代，不断调整和计算估计值，直到满足精度要求。
4. 在每次迭代中，构建量子电路并运行测量，计算测量结果的概率，并更新置信区间和 `theta` 区间。
5. 最终，计算出估计值及其置信区间，并将这些结果存储在 `IterativeAmplitudeEstimationResult` 对象中返回。

3.12 返回Delta测试结果

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

在这段代码中，首先将 `result_delta.confidence_interval_processed` 转换为一个 NumPy 数组，并将其赋值给变量 `conf_int`。

`result_delta.confidence_interval_processed` 是一个包含置信区间的对象，通过使用 `np.array` 函数将其转换为 NumPy 数组，可以方便地进行数值计算和操作。

接下来，代码使用 `print` 函数输出三个信息：

- 精确的 **Delta** 值:
- 估计的 **Delta** 值
- 置信区间 (**conf_int**)

4 实验测试

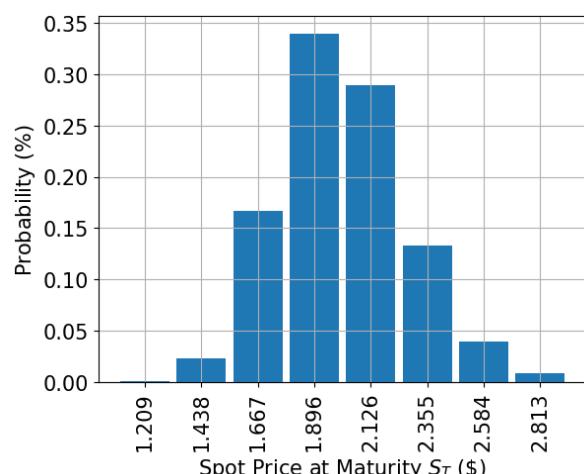
这里是实验测试

- 更换概率模型
 - 对数正态概率分布
 - 正态概率分布中选择
- 更换参数
 - 模型数据来源的量子比特数
 - 预期收益和Delta计算的精度 ϵ
 - 预期收益和Delta计算的置信度 α
 - 标的资产的当前价格 S_0
 - 无风险利率 r
 - 波动率 v
 - 到期时间 T

4.1 测试不同概率模型

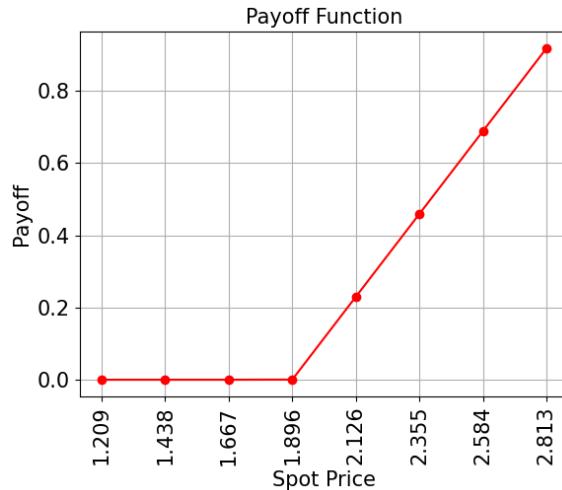
4.1.1 对数正态概率分布

不确定模型的概率分布:



精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)



测得结果：

```
# 不确定模型的量子比特数"num_uncertainty_qubits"，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = mean - 3 * stddev
high = mean + 3 * stddev
```

```
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: 0.1675
Confidence interval: [0.1611, 0.1738]
```

```
# 设置期权的执行价格(应该在不确定性的最低值和最高值之间)
strike_price = 1.896

# 设置目标数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(
```

```
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: 0.8898
Estimated value: 0.8898
Confidence interval: [0.8830, 0.8144]
```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格：2.0
- 波动率：0.4
- 无风险利率：0.05
- 期权到期时间：40/365

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格：1.896

3. 左上角表示期望收益

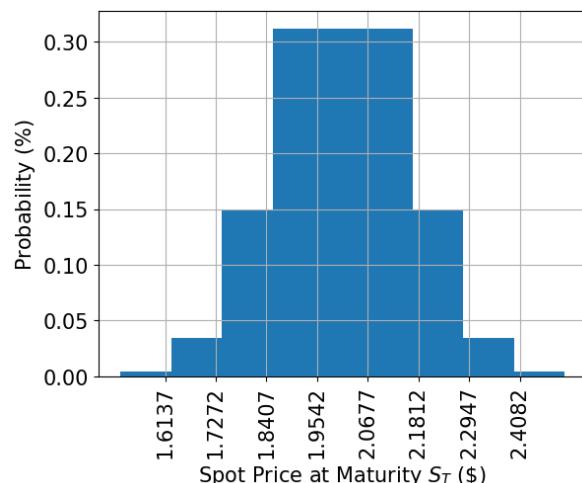
- 精确期望：0.1623
- 模型预估期望：0.1675
- 置信区间：[0.1611, 0.1738]

4. 右下角表示Delta

- 精确Delta：0.8098
- 模型预估Delta：0.8087
- 置信区间：[0.8030, 0.8144]

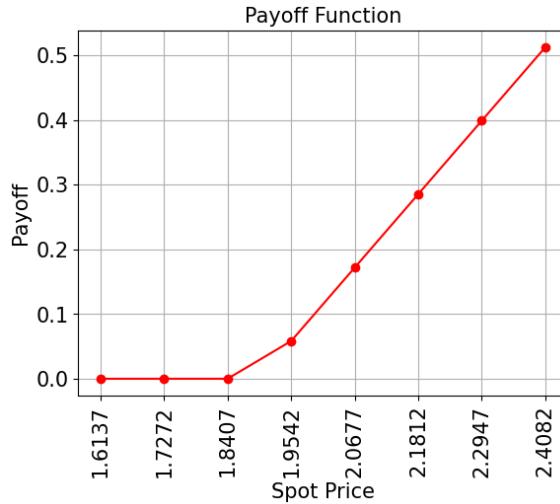
4.1.2 正态概率分布

不确定模型的概率分布：



精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)



测得结果：

```

# task.ipynb U <-- lognormal.py
# 不确定模型的量子比特数'num_uncertainty_qubits', 这个值越大, 精度越高, 但是计算量也越大
# 期望收益量子电路构建 > M+ 执行预估期望收益的量子计算 > conf_int = np.array(result.confidence_interval_processed)
# 生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
# 不确定模型的量子比特数'num_uncertainty_qubits', 这个值越大, 精度越高, 但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(老练的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)
# 为了计算方便, 我们将股票价格的最低值和最高值设置为均值的3倍标准差
[36] 1.9s
... C:\Users\16579\AppData\Local\Temp\ipykernel_8996\293918330.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "seed": 100}) is deprecated, use epsilon, alpha, and sampler instead
conf_int = np.array(result.confidence_interval_processed)
print("Exact value: " + str(exact_value))
print("Estimated value: " + str(result.estimation_processed))
print("Confidence interval: " + str(tuple(conf_int)))
[37] 0.0s
... Exact value: 0.1301
Estimated value: 0.1723
Confidence interval: [0.1664, 0.1783]
Python

# 任务对三角式的易操作性, 我们近似了期望收益的任务。
[38] 0.0s
... summer-camp-2023 > task.ipynb ...
summer-camp-2023 > task.ipynb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
[39] 0.0s
... conf_int = np.array(result_delta.confidence_interval_processed)
print("Exact delta: " + str(exact_delta))
print("Estimated value: " + str(european_call_delta.interpret(result_delta)))
print("Confidence interval: " + str(tuple(conf_int)))
[40] 0.0s
... Exact delta: 0.8121
Estimated value: 0.8108
Confidence interval: [0.8051, 0.8165]
Python

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格： 2.0
- 波动率： 0.4
- 无风险利率： 0.05
- 期权到期时间： 40/365

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格： 1.896

3. 左上角表示期望收益

- 精确期望： 0.1301

- 模型预估期望: 0.1723
- 置信区间: [0.1664, 0.1783]

4. 右下角表示Delta

- 精确Delta: 0.8121
- 模型预估Delta: 0.8108
- 置信区间: [0.8051, 0.8165]

4.1.3 结果分析

参考资料:

[对数正态分布和正态分布 \(lognormal vs normal\) 波动率的区别 - 知乎 \(zhihu.com\)](#)

可以从结果看到:

在预期收益的计算中，对数正态分布模型更接近其精确值。

在Delta的计算中，两者模型的精确计算能力接近。

我们这里简单聊一下为什么对数正态模型更适合欧式看涨期权定价问题：

- 对数正态分布 - 指数增长，取值必须为正，具有较大偏度，用于模拟资产价格的长期趋势。
- 正态分布 - 稳定增长，取值可以为任意实数，呈对称分布，用于描述价格的短期波动，表现为波动率。（尽管我们将正态分布移动到了正数部分并根据对应金融参数计算，但是效果依然不是很理想）

在金融市场中，对数正态分布模型通常被认为更适用于衍生品（如期权）的定价，因为它能够更好地描述资产价格的概率分布。

- 在金融市场中，资产价格通常被假设为对数正态分布。这是因为资产价格不能为负，而对数正态分布的取值范围是正数，符合这一特性。
- Black-Scholes模型基于资产价格的对数正态分布假设，推导出了欧式期权的定价公式。该模型在理论和实践中得到了广泛应用，并且与市场数据有较好的拟合。

而正态分布模型波动率则常用于固定收益类产品和利率衍生品的分析。

- 正态分布模型在期权定价中的应用较少，因为它无法很好地处理资产价格的非负性问题。

因此在欧式看涨期权的定价中，对数正态概率分布模型通常被认为是更优的选择。

4.2 测试相同概率模型不同参数下的定价结果

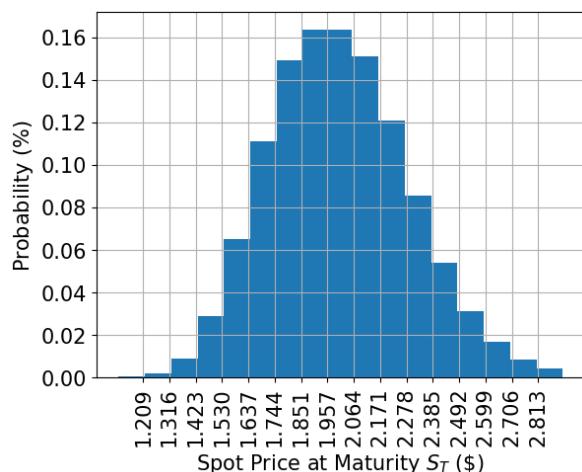
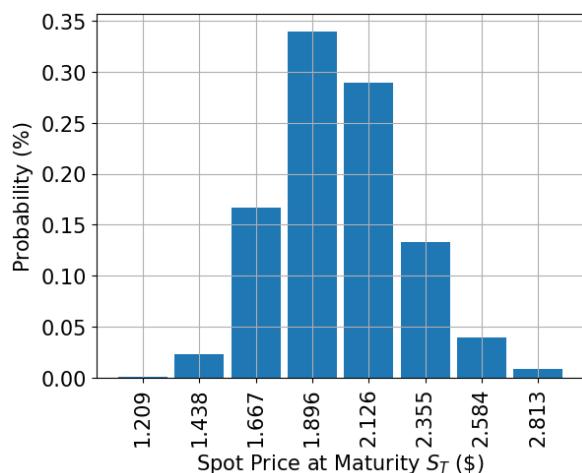
这里可以一并探究算法中参数对最终结果的影响

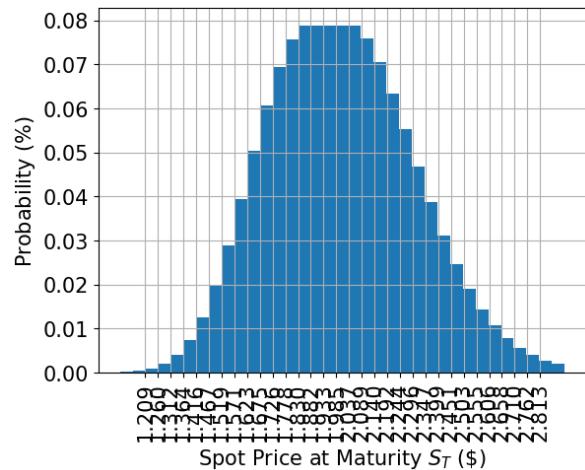
*我们这里选择对数正态概率分布模型

4.2.1 模型数据来源的量子比特数

我们选择3、4、5作为测试参数

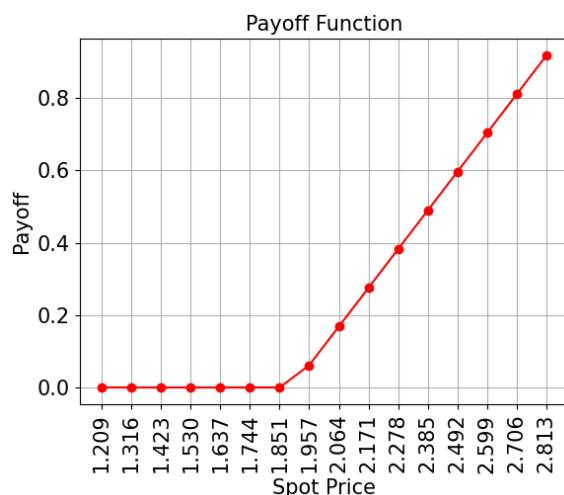
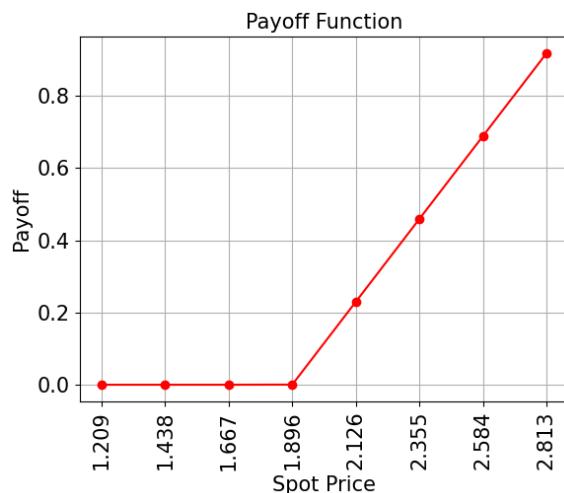
不确定模型的概率分布：

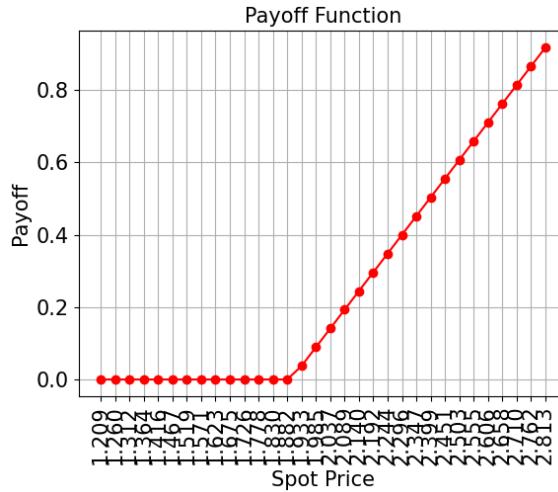




精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)





测得结果：

```

[43] taskipyrb U x lognormal.py ...
分布下的不确定模型 > # 不确定模型的量子比特数'num_uncertainty_qubits', 这个值越大, 精度越高, 但是计算量也越大
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 不确定模型的量子比特数'num_uncertainty_qubits', 这个值越大, 精度越高, 但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 对数正态分布得到的参数
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)
# 为了计算方便, 我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = mean + 3 * stddev

[44] taskipyrb U x ...
summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]

[45] taskipyrb U x lognormal.py ...
summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 不确定模型的量子比特数'num_uncertainty_qubits', 这个值越大, 精度越高, 但是计算量也越大
num_uncertainty_qubits = 4

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 对数正态分布得到的参数
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)
# 为了计算方便, 我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = mean + 3 * stddev

[46] taskipyrb U x ...
summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price

[47] taskipyrb U x ...
summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 期望收益量子电路构建 > 执行预估期望收益的量子计算 > conf_int = np.array(result.confidence_interval_processed)
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

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))

[48] 1.0s Python
C:\Users\16579\AppData\Local\Temp\ipykernel_8996\2393410330.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "seed": 1}) is deprecated, use epsilon=epsilon, alpha=alpha, sampler=RefSampler instead
  conf_int = np.array(result.confidence_interval_processed)
...
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))

[49] 0.0s Python
Exact value: 0.1623
Estimated value: 0.1675
Confidence interval: [0.1611, 0.1738]

> 测量Delta的电路构建

[50] taskipyrb U x ...
summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 期望收益量子电路构建 > 执行预估期望收益的量子计算 > conf_int = np.array(result.confidence_interval_processed)
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

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))

[51] 0.0s Python
Exact delta: 0.8098
Estimated value: 0.8087
Confidence interval: [0.0030, 0.8140]

[52] taskipyrb U x ...
summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 期望收益量子电路构建 > 执行预估期望收益的量子计算 > conf_int = np.array(result.confidence_interval_processed)
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

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))

[53] 3.1s Python
C:\Users\16579\AppData\Local\Temp\ipykernel_8996\2393410330.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "seed": 1}) is deprecated, use epsilon=epsilon, alpha=alpha, sampler=RefSampler instead
  conf_int = np.array(result.confidence_interval_processed)
...
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))

[54] 0.0s Python
Exact value: 0.1686
Estimated value: 0.1728
Confidence interval: [0.1669, 0.1788]

[55] taskipyrb U x ...
summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

# 期望收益量子电路构建 > 执行预估期望收益的量子计算 > conf_int = np.array(result.confidence_interval_processed)
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

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))

[56] 0.0s Python
Exact delta: 0.6351
Estimated value: 0.6357
Confidence interval: [0.6327, 0.6387]

```

```

# 不确定模型的量子比特数 num_uncertainty_qubits，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 5

# 期权参数(标的的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差

```

```

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置置信度的近似缩放
c_approx = 0.25

# 选择分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = linearAmplitudeFunction

```

```

conf_int = np.array(result.confidence_interval_processed)
print("Exact value: %.4f" % exact_value)
print("Estimated value: %.4f" % (result.estimate_processed))
print("Confidence interval: [%d, %d]" % tuple(conf_int))

Exact value: 0.1673
Estimated value: 0.1718
Confidence interval: [0.1659, 0.1778]

```

```

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

Exact delta: 0.6292
Estimated value: 0.6298
Confidence interval: [0.6275, 0.6320]

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格: **2.0**
- 波动率: **0.4**
- 无风险利率: **0.05**
- 期权到期时间: **40/365**

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格: **1.896**

3. 左上角表示期望收益

- 精确期望: **0.1623→0.1686→0.1673**
- 模型预估期望: **0.1675→0.1728→0.1718**
- 置信区间: **[0.1611, 0.1738]→[0.1669, 0.1788]→[0.1659, 0.1778]**

4. 右下角表示Delta

- 精确Delta: **0.8098→0.6351→0.6292**
- 模型预估Delta: **0.8087→0.6357→0.6298**
- 置信区间: **[0.8030, 0.8144]→[0.6327, 0.6387]→[0.6275, 0.6320]**

结果分析:

- 当量子比特数增多时, 期望收益偏差变小, 测量精度更高

- 当量子比特数增多时，**Delta**值预估偏差变大，但变化幅度不大，测量精度有略微损失

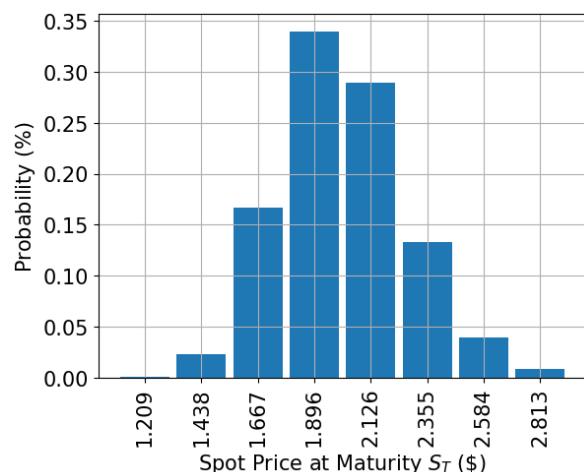
参数分析：

- 量子比特数量决定了离散化网格点的数量，即 2^n 个点。更多的量子比特意味着更高的离散化精度，从而更准确地表示对数正态分布。
- 振幅估计的精度与量子比特数量 n 有关。更多的量子比特可以提高振幅估计的精度，从而更准确地估计预期收益和 **Delta**。
- 更多量子比特：**
 - 更高的离散化精度，更准确地表示对数正态分布。
 - 振幅估计的精度提高，预期收益和**Delta**的估计更准确。

4.2.2 预期收益和**Delta**计算的精度 ϵ

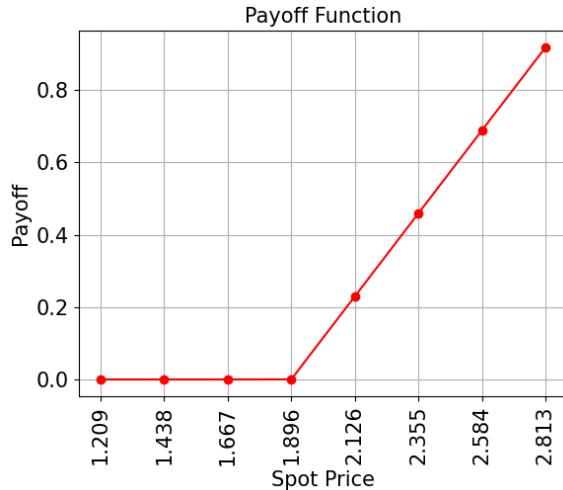
我们选择0.01、0.001、0.0001作为测试参数

不确定模型的概率分布：



精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)



测得结果：

不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[24] 2.1s
C:\Users\16579\AppData\Local\Temp\ipykernel_8996\2293181320.py:14: DeprecationWarning: The c
epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "seed": 75})
```

```
[25]
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))
```

```
[26] 0.0s
...
Exact value: 0.1623
Estimated value: 0.1675
Confidence interval: [0.1611, 0.1738]
```

Python

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

子电路对三类式的易操作性，我们近似了期望收益的任务。

```
[19]
# 设置期权的执行价格(应该在不确定性的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数 ...
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(
```

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[20]
# 不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
```

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

子电路对三类式的易操作性，我们近似了期望收益的任务。

```
[21]
# 设置期权的执行价格(应该在不确定性的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数 ...
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(
```

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[22]
# 不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
```

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[23]
# 不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
```

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[24] 94s
C:\Users\16579\AppData\Local\Temp\ipykernel_8996\324704858.py:14: DeprecationWarning: The c
epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "seed": 75})
```

```
[25]
conf_int = np.array(result.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))
```

```
[26] 0.0s
...
Exact delta: 0.8098
Estimated value: 0.8087
Confidence interval: [0.8030, 0.8140]
```

Python

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[27]
# 不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
```

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[28]
# 不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
```

taskipyb U x lognormal.py

summer-camp-2023 > taskipyb > ...

生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[29] 3.3s
C:\Users\16579\AppData\Local\Temp\ipykernel_8996\1362956927.py:9: DeprecationWarning: The c
epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "seed": 75})
```

```
[30]
conf_int = np.array(result.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))
```

```
[31] 0.0s
...
Exact delta: 0.8098
Estimated value: 0.8099
Confidence interval: [0.8093, 0.8105]
```

Python

```

# 不确定模型的量子比特数'num_uncertainty_qubits',这个值越大,精度越高,但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 对数正态分布得到的参数
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)

# 电路对三角式的易操作性,我们从上到下逐层进行操作。
epsilon = 0.01

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = linearAmplitudeFunction

```

```

[108] 33.4s
... C:\Users\16579\AppData\Local\Temp\ipykernel_8996\3948259583.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSamplerOptions={'shots': 100, "...
... conf_int = np.array(result.delta.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))

[109] 0.0s
... Exact value: 0.1623
Estimated value: 0.1687
Confidence interval: [0.1687, 0.1688]

[110] 0.0s
... C:\Users\16579\AppData\Local\Temp\ipykernel_8996\3145971946.py:9: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSamplerOptions={'shots': 100, "...
... 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))

[111] 0.0s
... Exact delta: 0.8098
Estimated value: 0.8098
Confidence interval: [0.8098, 0.8098]

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格: **2.0**
- 波动率: **0.4**
- 无风险利率: **0.05**
- 期权到期时间: **40/365**

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格: **1.896**

3. 左上角表示期望收益

- 精确期望: **0.1623**
- 模型预估期望: **0.1684→0.1684→0.1687**
- 置信区间: **[0.1611, 0.1738]→[0.1671, 0.1697]→[0.1687, 0.1688]**

4. 右下角表示Delta

- 精确Delta: **0.8098**
- 模型预估Delta: **0.8087→0.8099→0.8098**
- 置信区间: **[0.8030, 0.8144]→[0.8093, 0.8105]→[0.8098, 0.8098]**

结果分析:

- 当精度值越来越小(越来越精确)时, 期望收益偏差变小, 测量精度更高
- 当精度值越来越小(越来越精确)时, Delta值预估偏差变小, 甚至当 ϵ 为0.0001时, Delta值与精确值一致, 误差极小

参数分析:

- ϵ 是振幅估计算法的目标精度, 表示估计结果与真实值之间的最大误差。
- 振幅估计的目标是估计一个概率 p , 使得估计值 \hat{p} 满足:

$$|\hat{p} - p| \leq \epsilon$$

- 振幅估计的计算复杂度与 $1/\epsilon$ 成正比。
- ϵ 越小, 估计的精度越高, 但所需的计算资源 (如量子比特数量和量子门操作次数) 也会增加。

ϵ 越小:

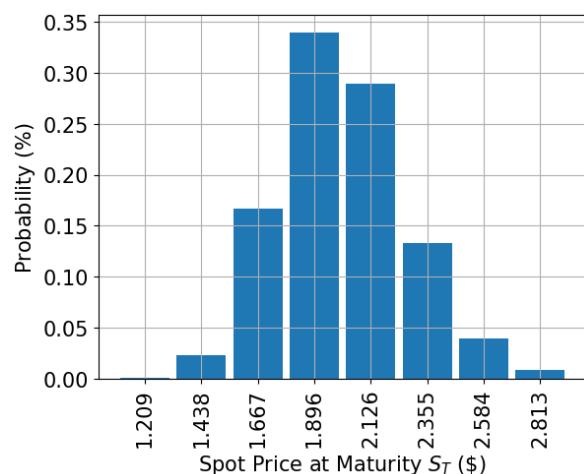
- 振幅估计的精度越高, 预期收益的估计误差越小。
- 振幅估计的精度越高, Delta 的估计误差越小

通过实验可以验证, ϵ 越小, 预期收益和 Delta 的估计精度越高, 但同时也会增加计算复杂度和资源消耗, 在这次运行中, 一个估计函数运行了大约4分钟, 内存达到占用90%。

4.2.3 预期收益和Delta计算的置信度 α

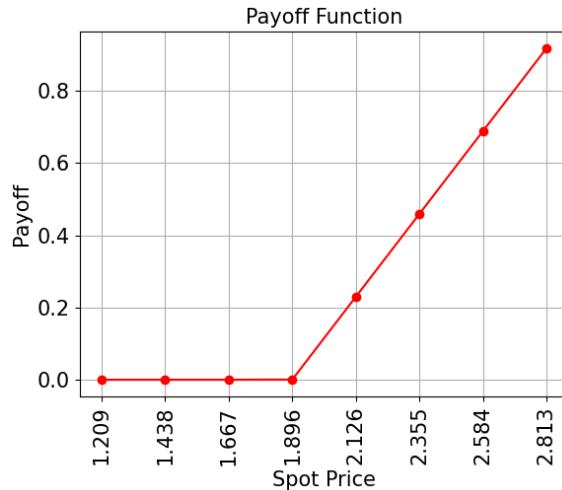
我们这里以5%、3%、1%作为测试数据

不确定模型的概率分布:



精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)



测得结果：

```
# 不确定模型的量子比特数"num_uncertainty_qubits",这个值越大,精度越高,但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便,我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = mean - 3 * stddev
high = mean + 3 * stddev
```

```
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: 0.1675
Confidence interval: [0.1611, 0.1738]
```

```
# 设置期权的执行价格(应该在不确定性的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(
```

```
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: 0.8898
Estimated value: 0.8898
Confidence interval: [0.8830, 0.8144]
```

```

# 不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也就越大
num_uncertainty_qubits = 3

# 期权参数(美元化的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 对对数正态分布进行采样
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)

# 为了计算方便，我们把股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = mean + 3 * stddev

```



```

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似值
c_approx = 0.25

# 设置分段线性回报函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = low
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,
)

```



```

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))

```



```

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))

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格: **2.0**
- 波动率: **0.4**
- 无风险利率: **0.05**
- 期权到期时间: **40/365**

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格: **1.896**

3. 左上角表示期望收益

- 精确期望: **0.1623**
- 模型预估期望: **0.1675→0.1674→0.1674**
- 置信区间: **[0.1611, 0.1738]→[0.1607, 0.1742]→[0.1599, 0.1749]**

4. 右下角表示Delta

- 精确Delta: **0.8098**
- 模型预估Delta: **0.8087→0.8105→0.8087**
- 置信区间: **[0.8030, 0.8144]→[0.8074, 0.8136]→[0.8020, 0.8154]**

结果分析:

- 当置信度 α 越来越小(置信区间越窄)时, 期望收益变化较小, 但是置信区间变大, 结果更可靠
- 当置信度 α 越来越小(置信区间越窄)时, **Delta**值预估偏差变小, 置信区间变大, 结果更可靠

参数分析:

- 置信度 α 是振幅估计算法中的关键参数, 用于控制估计结果的置信水平。
 - 例如, $\alpha = 0.05$ 表示估计结果的置信区间为 95%。
- 振幅估计的置信区间为:

$$[\hat{p} - z_{\alpha/2} \cdot \text{SE}, \hat{p} + z_{\alpha/2} \cdot \text{SE}]$$

其中:

- \hat{p} : 估计的概率值。
- $z_{\alpha/2}$: 标准正态分布的分位数 (如 $\alpha = 0.05$ 时, $z_{\alpha/2} = 1.96$)。
- SE: 估计的标准误差。
- α 越小:
 - 置信区间越窄, 预期收益和 **Delta** 的估计精度越高。
 - 但所需的计算资源 (如量子比特数量和量子门操作次数) 也会增加。

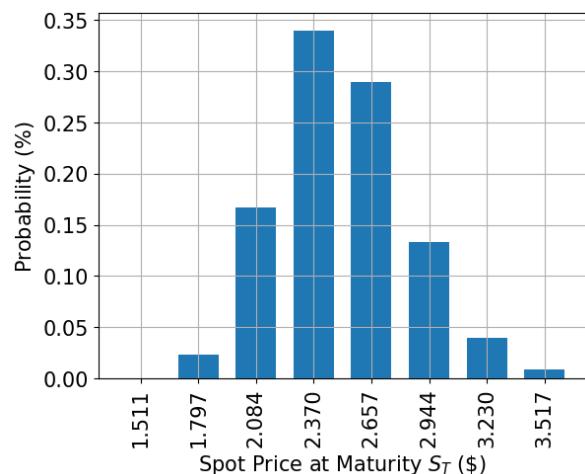
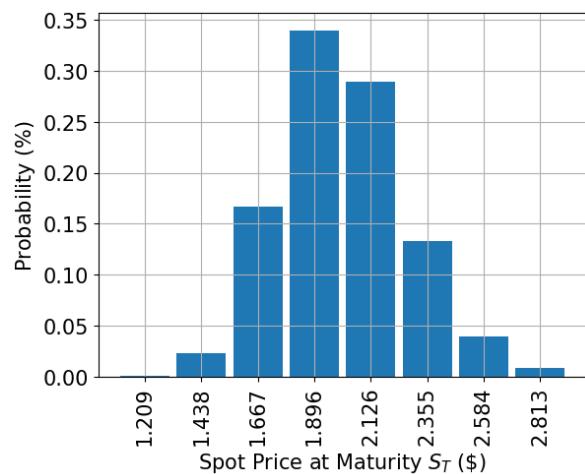
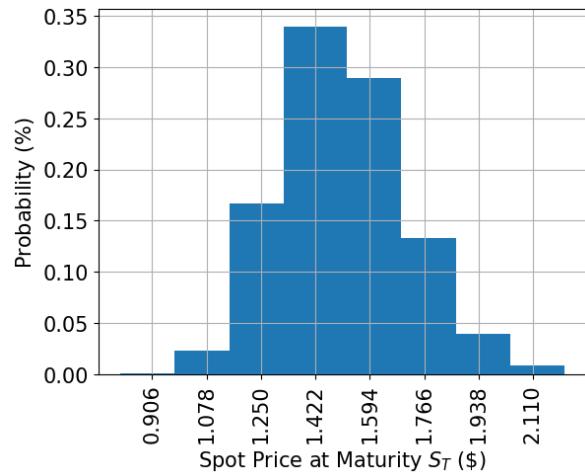
4.2.4 标的资产的当前价格 S_0

我们这里以1.5、2、2.5作为测试数据

参考:

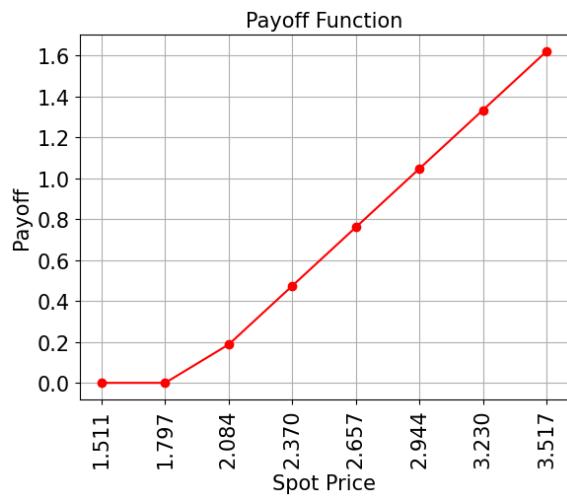
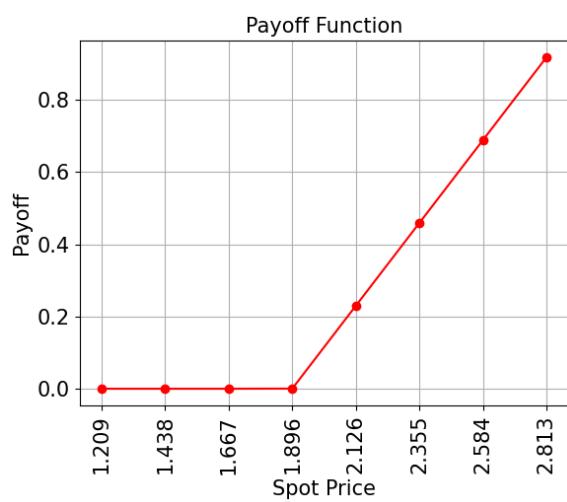
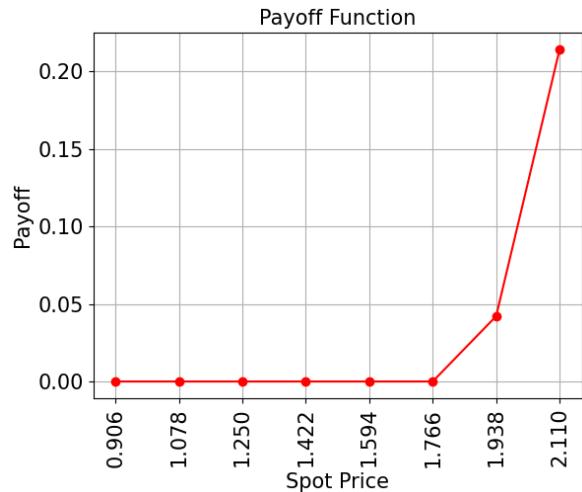
[在期权有效期内, 基础资产的变化如何影响期权价格? - 知乎 \(zhihu.com\)](#)

不确定模型的概率分布:



精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)



测得结果：

测量Delta的电路构建

```

# 不确定模型的量子比特数'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 1.5 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差

```

```

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]

```

```

# 不确定模型的量子比特数'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差

```

```

# 设置期权的执行价格(应该在不确定性的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(

```

```

# 不确定模型的量子比特数'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.5 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差

```

```

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格: $1.5 \rightarrow 2.0 \rightarrow 2.5$
- 波动率: 0.4
- 无风险利率: 0.05
- 期权到期时间: $40/365$

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格: 1.896

3. 左上角表示期望收益

- 精确期望: $0.0034 \rightarrow 0.1623 \rightarrow 0.6180$
- 模型预估期望: $0.0056 \rightarrow 0.1675 \rightarrow 0.6231$
- 置信区间: $[0.0035, 0.0077] \rightarrow [0.1611, 0.1738] \rightarrow [0.6068, 0.6393]$

4. 右下角表示Delta

- 精确Delta: $0.0476 \rightarrow 0.8098 \rightarrow 0.9766$
- 模型预估Delta: $0.0473 \rightarrow 0.8087 \rightarrow 0.9762$
- 置信区间: $[0.0460, 0.0485] \rightarrow [0.8030, 0.8144] \rightarrow [0.9745, 0.9779]$

结果分析:

- 当标的资产的当前价格越来越大时, 期望收益变大, 意味着期权的收益较高, 盈利较多
- 当标的资产的当前价格越来越大时, Delta值越来越大, 表示金融风险变大

参数分析:

- S_0 : 标的资产在当前时刻的价格。
 - S_0 越大, 越比行权价格 K 大, 标的资产价格 S_T 在到期日超过 K 的概率增加, 预期收益的值增大。
 - S_0 越大, 越比行权价格 K 大, 标的资产价格 S_T 在到期日超过 K 的概率增加, Delta 的值增大。

期权定价的最基本思想是无套利原则。对欧式看涨期权而言, 一般地, 有

$$C(S, t) = S_0 N(d_1) - K e^{-rT} N(d_2)$$

很明显，当 $S_0 > K$ 时，基础资产价格上升会使得期权价格上升。

以最常见的 **Black-Scholes** 期权定价模型进行扩展说明。对于欧式看涨期权而言，模型公式如下：

$$C(S, t) = S_0 N(d_1) - K e^{-rT} N(d_2)$$

其中

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$
$$d_2 = d_1 - \sigma\sqrt{T}$$

$$\Delta = \frac{\partial C}{\partial S_0} = N(d_1)$$

Δ 是标的资产价格对期权价格的一阶偏导，衡量的是基础资产价格对影响期权价格的影响。学术上，一般称之为 **Delta**。

一般地，当不存在红利支付时，欧式看涨期权的 **Delta** 为：

$$N(d_1)$$

所以对欧式看涨期权而言，基础资产价格与期权价格正相关；对欧式看跌期权而言，基础资产价格和期权价格负相关。

4.2.5 波动率 v

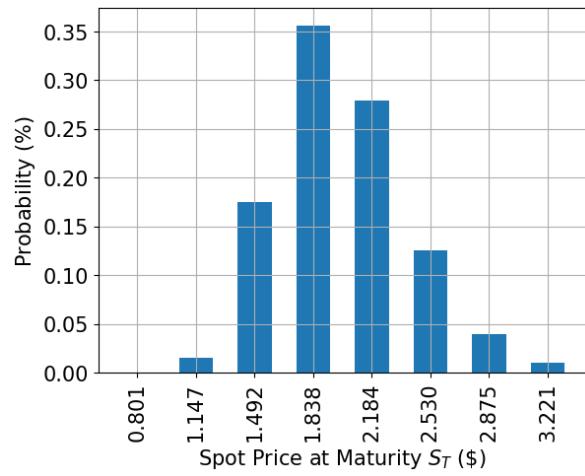
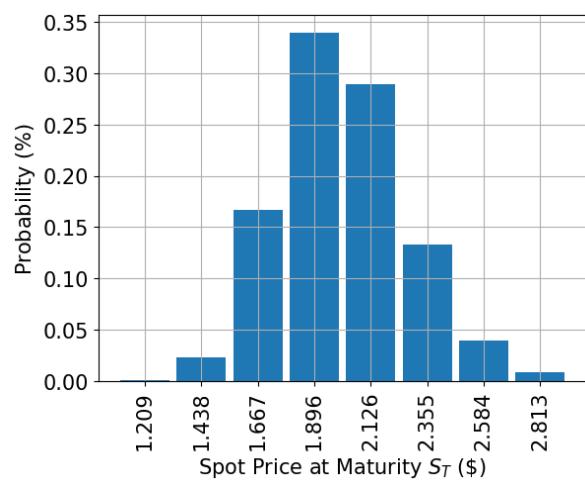
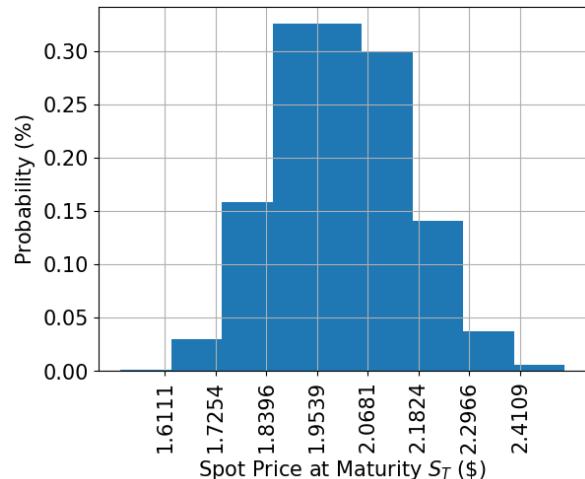
我们这里以 0.2、0.4、0.6 作为测试数据

参考：

[波动率如何影响期权的价值？ - 知乎 \(zhihu.com\)](#)

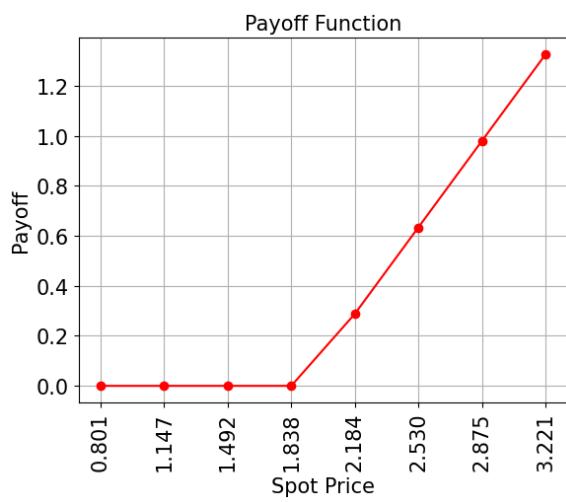
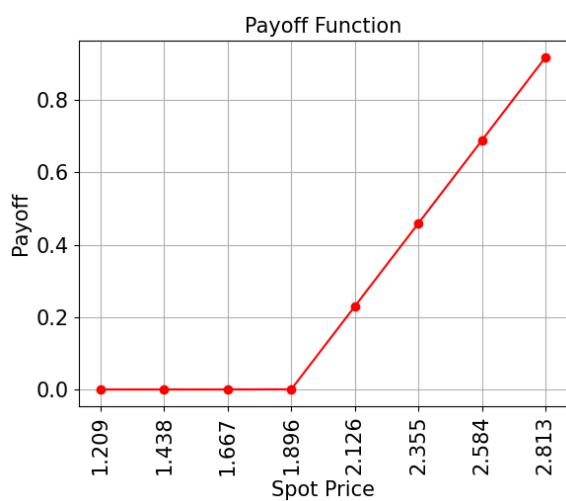
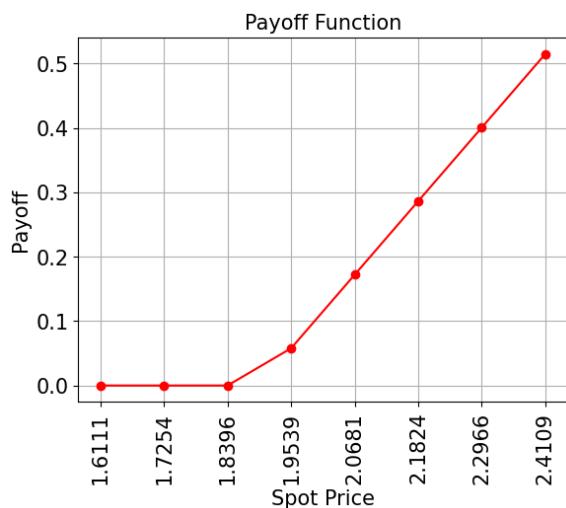
[到期时间、波动率对Delta的影响以及实际应用](#)

不确定模型的概率分布：



精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)



测得结果：

The screenshot shows a Jupyter Notebook interface with several code cells. The first cell calculates the price of a call option under different uncertainty models:

```
# 不确定模型的量子比特数'num_uncertainty_qubits', 这个值越大, 精度越高, 但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.2 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便, 我们将股票价格的最低值和最高值设置为均值的3倍标准差
```

The second cell calculates the confidence interval for the result:

```
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))
[50] ✓ 0.0s
```

The third cell calculates the option price under a Delta model:

```
# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(
```

The fourth cell calculates the confidence interval for the result:

```
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))
[54] ✓ 0.0s
```

task.ipynb x lognormal.py

不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
# 不确定模型的量子比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了方便理解。我们将股票价格的最低值和最高值设置为均值的3倍标准差
[19]
```

task.ipynb x

summer-camp-2023 > task.ipynb > ...

生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

| 电子路对三角角式的易操作性，我们近似了期望收益的任务。

```
# 设置期权的执行价格(应该在不确定性的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = linearAmplitudeFunction[20]
```

task.ipynb x

normal.py uniform.py linear_amplitude.function.py task.ipynb x

summer-camp-2023 > task.ipynb > ... 期望收益量子电路构建

生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
[24] ✓ 2.1s Python
```

```
C:\Users\16579\AppData\Local\Temp\ipykernel_8996\2392\i18338.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "seed": 1}) is deprecated, use qiskit.algorithms.QAOA instead.
```

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

[25] ✓ 0.0s Python

```
Exact value: 0.1623
Estimated value: 0.1675
Confidence interval: [0.1611, 0.1738]
```

task.ipynb x

summer-camp-2023 > task.ipynb > ...

生成 + 代码 + Markdown | 全部运行 ⏪ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

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

[26] ✓ 0.0s Python

```
Exact delta: 0.8098
Estimated value: 0.8087
Confidence interval: [0.0830, 0.8144]
```

```

1. 对数正态分布
2. 正态分布

# 不确定模型的量子比特数'num_uncertainty_qubits', 这个值越大, 精度越高, 但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.6 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
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,
)

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格: **2.0**
- 波动率: **0.4**
- 无风险利率: **0.05**
- 期权到期时间: **40/365**

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格: **1.896**

3. 左上角表示期望收益

- 精确期望: **0.1291→0.1623→0.2124**
- 模型预估期望: **0.1288→0.1675→0.2202**
- 置信区间: **[0.1169, 0.1406]→[0.1611, 0.1738]→[0.2107, 0.2298]**

4. 右下角表示Delta

- 精确Delta: **0.8103→0.8098→0.4548**
- 模型预估Delta: **0.8091→0.8087→0.4557**
- 置信区间: **[0.8034, 0.8148]→[0.8030, 0.8144]→[0.4518, 0.4597]**

结果分析:

- 当波动率越高时，期望收益变大，意味着期权的收益较高，盈利较多
- 当波动率越高时，**Delta**值越来越小，表示金融风险变小

参数分析：

- 对于期权来说，波动率越高，盈利性越高；波动率越低，盈利性越低

一般来说，波动率与期权价格呈正向关系。无论是认购期权还是认沽期权，在其他条件不变的情况下，波动率越高，期权价格越高。这是因为较高的波动率意味着标的资产价格在期权剩余期限内出现大幅波动的可能性更大，从而使得期权到期时处于实值状态（对于认购期权是标的资产价格高于行权价，对于认沽期权是标的资产价格低于行权价）的概率增加，期权的价值也就相应提高

- **Delta**值表示的是标的物价格变化与期权价格变化的关系，实值期权的**Delta**绝对值与波动率负相关

波动率是标的资产价格的年化标准差，表示标的资产价格的不确定性。波动率越高，标的资产价格的变化幅度越大；波动率越低，标的资产价格的变化幅度越小。

波动率越高：

- 标的资产价格 S_T 的不确定性增加，Delta 的值增大。

在 Black-Scholes 模型中，Delta 的计算公式为：

$$\Delta = N(d_1)$$

其中：

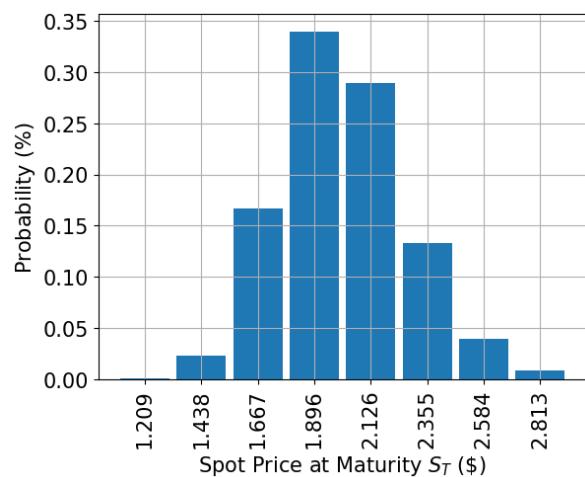
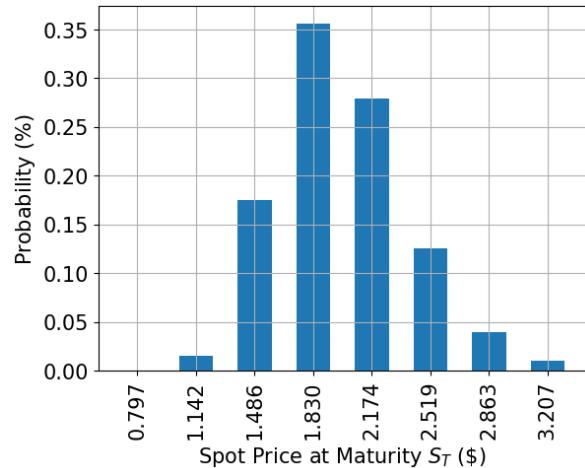
$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

- 波动率 σ 出现在 d_1 的分子分母中，根据欧式看涨期权相关参数的限制，我们可以推知Delta与波动率负相关

4.2.6 无风险利率 r

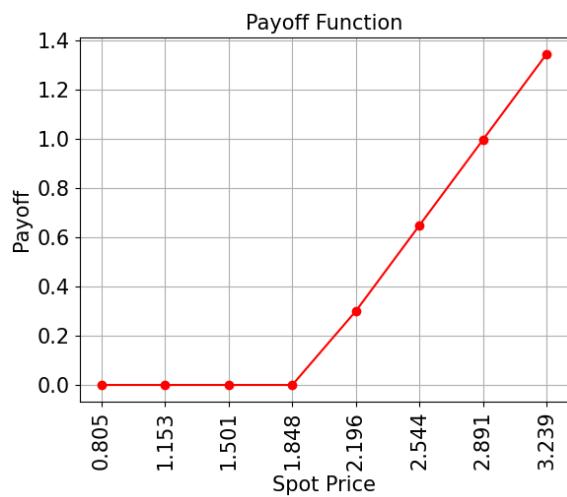
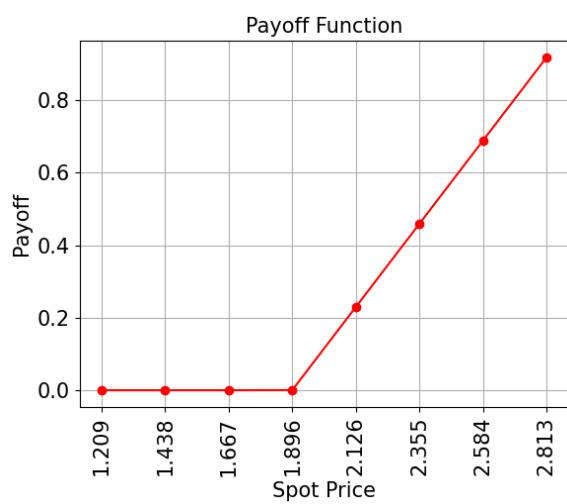
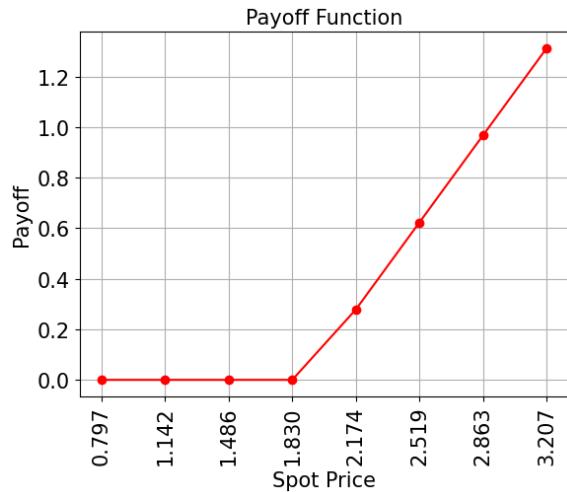
我们这里以0.01、0.05、0.10作为测试数据

不确定模型的概率分布：



精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)



测得结果：

camp-2023 > task.ipynb > 欧式看涨期权定价 > 期初不確定模型分布(RN-模型) > 画出不確定模型的圖表分布 > M4 期初不确定量子子的值 > conf_int = np.array(result.confidence_interval_processed)

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
# 不确定模型的量子比特数'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始股票价格
vol = 0.6 # 波动率
r = 0.01 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = np.minimum(100, mean + 3 * stddev)
```

[88] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
```

[19] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
# 不确定模型的量子比特数'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
```

[19] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
# 设置期权的执行价格(应该在不确定性的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = linearAmplitudeFunction
```

[78] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
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))
```

[78] ✓ 0s Python

[78] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
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))
```

[78] ✓ 0s Python

[78] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
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))
```

[78] ✓ 0s Python

[78] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
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))
```

[78] ✓ 0s Python

[78] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
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))
```

[78] ✓ 0s Python

[78] task.ipynb U x

生成 + 代码 + Markdown | 全部运行 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```
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))
```

[78] ✓ 0s Python

```

# 不确定模型的量化比特数 'num_uncertainty_qubits'，这个值越大，精度越高，但是计算量也越大
num_uncertainty_qubits = 3

# 期权的参数(考虑的随机分布得到的参数)
S = 2.0 # 初始化股票价格
vol = 0.6 # 波动率
r = 0.10 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)
# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = np.minimum(10, mean + 3 * stddev)

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
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,
)

```

```

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: 0.2183
Estimated value: 0.2266
Confidence interval: [0.2188, 0.2343]

```

```

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: 0.4548
Estimated value: 0.4557
Confidence interval: [0.4518, 0.4597]

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格: **2.0**
- 波动率: **0.4**
- 无风险利率: **0.01→0.05→0.10**
- 期权到期时间: **40/365**

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格: **1.896**

3. 左上角表示期望收益

- 精确期望: **0.2077→0.1623→0.2183**
- 模型预估期望: **0.2152→0.1675→0.2266**
- 置信区间: **[0.2057, 0.2247]→[0.1611, 0.1738]→[0.2188, 0.2343]**

4. 右下角表示Delta

- 精确Delta: **0.4548→0.8098→0.4548**
- 模型预估Delta: **0.4557→0.8087→0.4557**
- 置信区间: **[0.4518, 0.4597]→[0.8030, 0.8144]→[0.4518, 0.4597]**

结果分析:

- 当无风险利率越来越大时，期望收益先变小后变大
- 当无风险利率越来越大时，**Delta**值先变大后变小

参数分析：

1. 期望收益

提高无风险利率会导致期权收益减少。这是因为——无风险利率的提高意味着投资者可以获得更高的无风险收益，因此对期权的投资需求将减少，导致期权价格下跌。但是由于标的资产的时间价值一直在增长，因此在这里的期望收益变化并不是成正/负相关的。我们通过数学式解释为什么会有上述变化：

无风险利率：无风险利率是指投资者可以获得的没有任何风险的收益率，通常用国债收益率来表示。

在期权定价中，无风险利率用于折现未来现金流，影响期权的现值。

- 欧式看涨期权的期望收益是期权在到期日的平均收益，即：

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

其中：

- S_T : 标的资产在到期日的价格。
- K : 行权价格。
- 在 **Black-Scholes** 模型中，欧式看涨期权的期望收益可以通过以下公式计算：

$$\mathbb{E} [\max\{S_T - K, 0\}] = S_0 e^{rT} N(d_1) - K N(d_2)$$

- 其中：
- S_0 : 标的资产的当前价格。
- r : 无风险利率。
- T : 到期时间。
- σ : 标的资产的波动率。
- $N(d)$: 标准正态分布的累积分布函数。
- d_1 和 d_2 的定义如下：

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

无风险利率 r 出现在 d_1 和 d_2 的分子中。当 r 增大时, d_1 和 d_2 的值增大, $N(d_1)$ 和 $N(d_2)$ 的值也增大。

- 无风险利率 r 增大:

- $S_0 e^{rT} N(d_1)$ 增大, 因为 e^{rT} 随 r 增大而增大。
- $KN(d_2)$ 增大, 因为 $N(d_2)$ 随 r 增大而增大。
- 但是, $S_0 e^{rT} N(d_1)$ 的增长速度通常比 $KN(d_2)$ 的增长速度快, 因此期望收益先变小后变大。

2. Delta

当无风险利率逐渐增大时, 欧式看涨期权的 **Delta** 通常会先变大后变小

- **Delta** 是期权价格对标的资产价格的敏感性, 定义为:

$$\Delta = \frac{\partial C}{\partial S_0} = N(d_1)$$

其中:

- C : 期权的价格。
- S_0 : 标的资产的当前价格。
- $N(d_1)$: 标准正态分布的累积分布函数。

对于欧式看涨期权, Delta 的值通常在 $[0, 1]$ 之间。

- 在 **Black-Scholes** 模型中, d_1 的定义为:

$$d_1 = \frac{\ln\left(\frac{S_0}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

- 无风险利率 r 增大:

- d_1 的分子增大, d_1 的值增大。
- $N(d_1)$ 的值也增大, 因为 $N(d)$ 是单调递增函数。

- 但是，当 r 增大到一定程度后， d_1 的值会趋于稳定， $N(d_1)$ 的值也会趋于稳定，Delta 的值开始减小。

4.2.7 到期时间 T

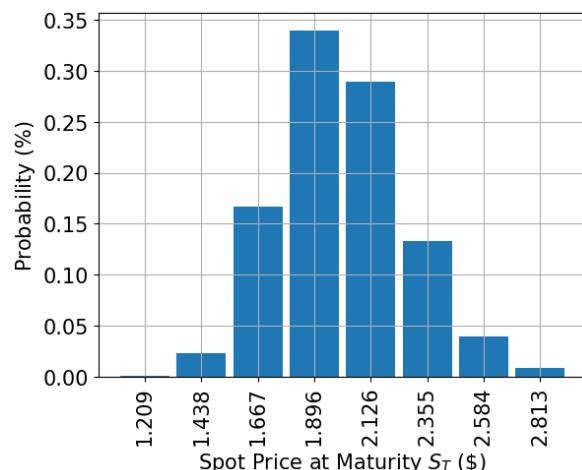
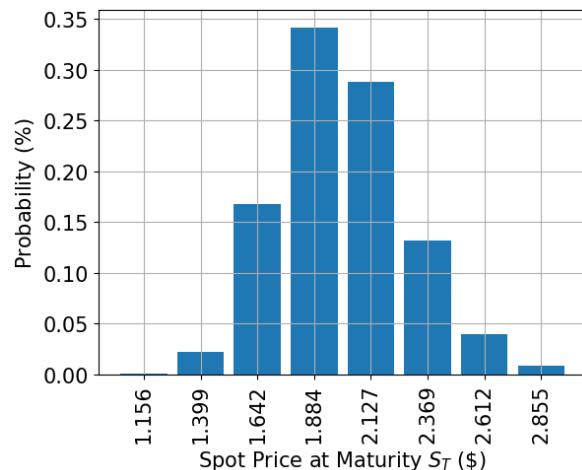
我们这里选择20、40、60作为到期时间测试数据

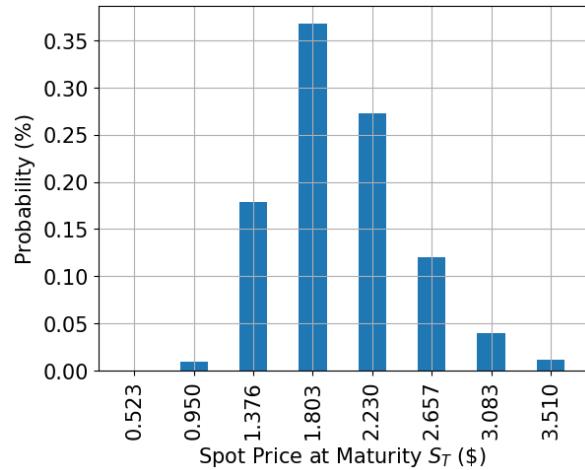
参考：

[为何到期期限对欧式看涨看跌期权的影响是“不一定”，而对美式的影响是正向？ - 高顿问答 \(gaodun.com\)](#)

[为什么对于美式期权来说，到期期限越长，价值越大？对于欧式期权来说，较长时间不一定能增加期权价值？ 百度知道 \(baidu.com\)](#)

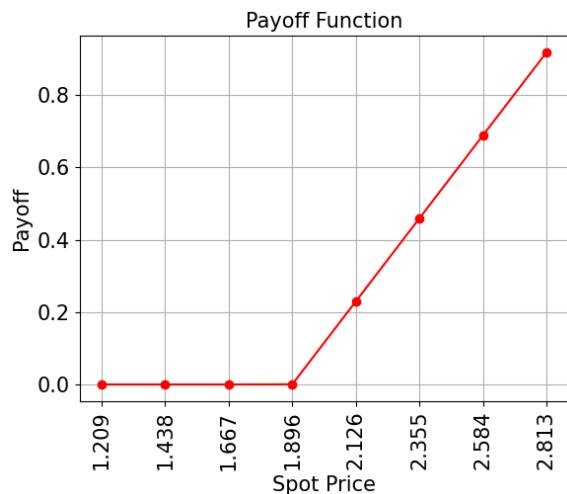
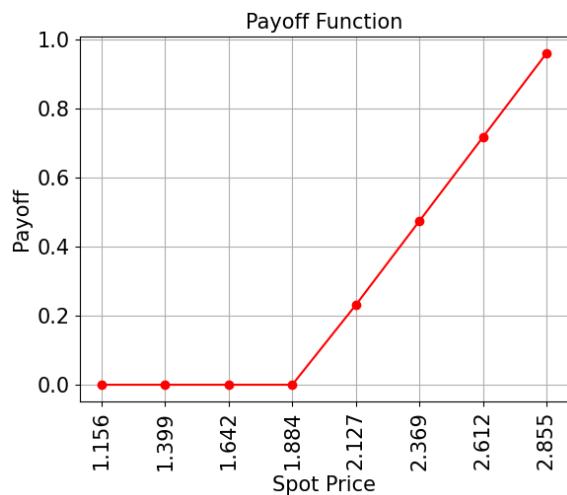
不确定模型的概率分布：

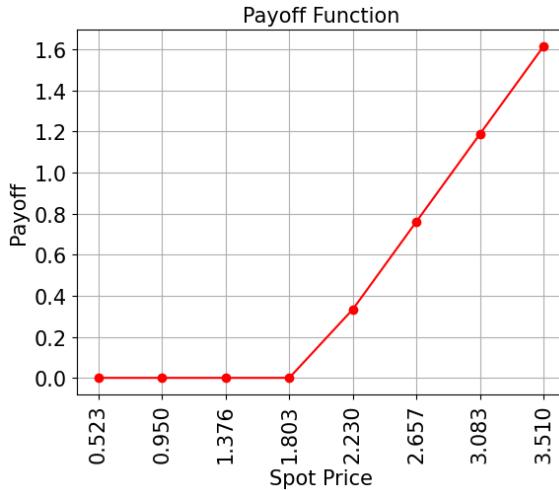




精确收益函数：

但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，执行价格附近最为明显)





测得结果：

```

# 分布下的不确定模型 > # 不确定模型的量子比特数 num_uncertainty_qubits，这个值越大，精度越高，但是计算量也越大 > M 执行预期收益量子电路构建 > M 执行预期期望收益的量子计算 > 
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
D < num_uncertainty_qubits = 3
# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.6 # 波动率
r = 0.05 # 无风险利率
T = 20 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = mean + 3 * stddev

```

```

[104] taskipyrb U ✘ 2.6s Python
... C:\Users\16579\AppData\Local\Temp\ipykernel_13092\23934\10338.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "...

```

```

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

```

```

[110] ✓ 0.0s Python
... Exact value: 0.1655
Estimated value: 0.1707
Confidence interval: [0.1640, 0.1773]

```

```

taskipyrb U ✘ ...

```

```

# 预期收益量子电路构建 > # 给定期权的精算收益函数(但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题，因此我们使用Delta的精算构建) > M 执行预期Delta的量子计算 > M 执行预期Delta的量子电路构建 > M 执行预期Delta的量子计算 > 
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
D < epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "...

```

```

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

```

```

[114] ✓ 0.0s Python
... Exact delta: 0.4682
Estimated value: 0.4688
Confidence interval: [0.4661, 0.4715]

```

```

taskipyrb U ✘ ...

```

```

# 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
breakpoints = [low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = high - strike_price
european_call_objective = LinearAmplitudeFunction(

```

```

taskipyrb U ✘ lognormal.py ...

```

```

# 分布下的不确定模型 > # 不确定模型的量子比特数 num_uncertainty_qubits，这个值越大，精度越高，但是计算量也越大 > M 执行预期收益量子电路构建 > M 执行预期期望收益量子电路构建 > 
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
D < num_uncertainty_qubits = 3
# 期权的参数(考虑的随机分布的参数)
S = 2.0 # 初始化股票价格
vol = 0.4 # 波动率
r = 0.05 # 无风险利率
T = 40 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便，我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = mean + 3 * stddev

```

```

[24] taskipyrb U ✘ 2.1s Python
... C:\Users\16579\AppData\Local\Temp\ipykernel_8996\23934\10338.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "...

```

```

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

```

```

[25] ✓ 0.0s Python
... Exact value: 0.1623
Estimated value: 0.1675
Confidence interval: [0.1611, 0.1738]

```

```

taskipyrb U ✘ ...

```

```

summer-camp-2023 > taskipyrb > ...
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)

```

```

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

```

```

[29] ✓ 0.0s Python
... Exact delta: 0.8098
Estimated value: 0.8087
Confidence interval: [0.8030, 0.8144]

```

```

# 分布下的不确定模型 > 不确定模型的量子比特数 num_uncertainty_qubits, 这个值越大, 精度越高, 但是计算量也越大 > M4 期望收益量子电路构建 > M4 执行预估期望收益的量子计算 > conf_int = np.array(result.confidence_interval_processed)
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
[116] num_uncertainty_qubits = 3
# 期权的参数(考虑的随机分布得到的参数)
S = 2.0 # 初始标股票价格
vol = 0.6 # 波动率
r = 0.05 # 无风险利率
T = 365 / 365 # 到期时间

# 由对数正态分布得到的参数
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)

# 为了计算方便, 我们将股票价格的最低值和最高值设置为均值的3倍标准差
low = np.maximum(0, mean - 3 * stddev)
high = mean + 3 * stddev

```



```

# 任务 ipynb U X ...
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
[117] # 绘制期权的精确收益函数(但是这里会由于量子比特数量过少造成离散点的斜率不对齐的问题, 拱...
生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16) 生成 + 代码 + Markdown | 全部运行 ⚡ 重启 ... summer-camp-qf-2023 (Python 3.10.16)
[118] # 设置期权的执行价格(应该在不确定模型的最低值和最高值之间)
strike_price = 1.896

# 设置目标函数的近似缩放
c_approx = 0.25

# 设置分段线性目标函数
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,
)
```



```

... C:\Users\16579\AppData\Local\Temp\ipykernel_13092\2393d10330.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "...
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))

```



```

... C:\Users\16579\AppData\Local\Temp\ipykernel_13092\2393d10330.py:14: DeprecationWarning: epsilon_target=epsilon, alpha=alpha, sampler=RefSampler(options={"shots": 100, "...
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))

```

1. 左上角代表金融模型相关数据

- 标的资产的当前价格: **2.0**
- 波动率: **0.4**
- 无风险利率: **0.05**
- 期权到期时间: **20/365→40/365→60/365**

2. 左下角代表当前投资人的期权的执行价格

- 期权的执行价格: **1.896**

3. 左上角表示期望收益

- 精确期望: **0.1655→0.1623→0.2481**
- 模型预估期望: **0.1707→0.1675→0.2583**
- 置信区间: **[0.1640, 0.1773]→[0.1611, 0.1738]→[0.2487, 0.2679]**

4. 右下角表示Delta

- 精确Delta: **0.4682→0.8098→0.4444**
- 模型预估Delta: **0.4688→0.8087→0.4453**
- 置信区间: **[0.4661, 0.4715]→[0.8030, 0.8144]→[0.4405, 0.4500]**

结果分析:

- 当到期时间越来越大时，期望收益先变小后变大
- 当到期时间越来越大时，**Delta**值先变大后变小

参数分析：

在欧式看涨期权的定价中，**到期时间 T** 是一个重要的参数，它对期权的期望收益和 **Delta**有显著影响。

在查阅相关资料后，我了解到对于欧式期权来说，期权持有者只能在到期日当天行权，故距离到期剩余时间长并不意味着到期日当天标的资产的价格对多头有利，因此，对于欧式期权来说，到期剩余时间对期权价格的影响具有不确定性。当股利存在时欧式看涨期权价值不一定随着到期时间增加而增加，因为可能增加到期时间后会把股利包含在期限内，因此导致期权时间价值下降。欧式期权合同要求其持有者只能在到期日履行合同，结算日是履约后的一天或两天。国内的外汇期权交易都是采用的欧式期权合同方式。

到期时间对欧式看涨期权的影响类似，也为不确定，变化方向与期望收益相反。

5 心得体会

1. 在配置**Qiskit-finance**时，发现原文档的命令发生了错误

```
1 | pip install "qiskit[finance]"  
  
➤ pip install "qiskit[finance]"  
Requirement already satisfied: qiskit[finance] in d:\conda\envs\summer-camp-qf-2023\lib\site-packages (1.3.1)  
WARNING: qiskit 1.3.1 does not provide the extra 'finance'
```

可以看到由于**Qiskit 1.3.1**版本后不再支持此条命令，我们通过以下文档进行修改

[qiskit.algorithms migration guide | IBM Quantum Documentation](#)

Algorithms migration guide

⚠ Caution

Deprecation Notice

This guide precedes the introduction of the V2 primitives interface. Following the introduction of the V2 primitives, some providers have deprecated V1 primitive implementations in favor of the V2 alternatives. If you are interested in following this guide, we recommend combining it with the [Migrate to V2 primitives](#) guide to bring your code to the most updated state.

In Qiskit 0.44 and later releases, the `qiskit.algorithms` module has been superseded by a new standalone library, `qiskit_algorithms`, available on [GitHub](#) and [PyPi](#). The `qiskit.algorithms` module was migrated to a separate package in order to clarify the purpose of Qiskit and make a distinction between the tools and libraries built on top of it.

If your code used `qiskit.algorithms`, follow these steps:

1. Check your code for any uses of the `qiskit.algorithms` module. If you are, follow this guide to migrate to the primitives-based implementation.
2. After updating your code, run `pip install qiskit-algorithms` and update your imports from `qiskit.algorithms` to `qiskit_algorithms`.

1. Check your code for any uses of the `qiskit.algorithms` module. If you are, follow this guide to migrate to the primitives-based implementation.
2. After updating your code, run `pip install qiskit-algorithms` and update your imports from `qiskit.algorithms` to `qiskit_algorithms`.

在切换pip安装命令后，将原文档引入的`qiskit.algorithms`换为`qiskit_algorithms`即可

2. 在执行部分文件时，发现部分采样器并不能正常工作，在开源社区中我发现有同样的问题出现，在issue下方我找到了解决方法，即利用Qiskit另外设计的采样器

[Segmentation fault when using Aer Sampler • Issue #2232 • Qiskit/qiskit-aer \(github.com\)](#)

[CI notebook tests failing with Segmentation fault in one of the notebooks \(kernel dies\) Issue #345](#)

最后用 Reference Sampler 替掉 Aer Sampler `

3. 在部分量子社区文档中，我发现了部分不再支持维护的代码，这可能是由于 Qiskit 工具更新工作交给 IBM 公司维护之后，原版本的代码没有更新的必要了

Qiskit Textbook content

This repository contains the source files for the Qiskit Textbook. Each page in the textbook is a Jupyter notebook in the [notebooks](#) folder.

Important

The Qiskit Textbook has been superseded by [IBM Quantum Learning](#). These source files are no longer maintained and may contain errors. We are not accepting any contributions.

4. 在上上个学期学完宏观经济学又补充学习了欧式看涨期权，又学到了很多(x

[期权定价中的行权概率该如何理解？](#)

6 选做任务(Bonus)

期权的价格分布一般通过训练量子生成对抗网络 (qGAN) 制备，请实现这一过程。

量子机器学习算法，即量子生成对抗网络 (**qGAN**) 来辅助欧式看涨期权的定价。更具体地说，可以训练 **qGAN**，以便量子电路对欧洲看涨期权标的资产的现货价格进行建模。然后，可以将生成的模型集成到基于量子振幅估计的算法中，以评估预期收益。

这里的重点在于我们要利用**qGAN**来处理量子比特的振幅生成，不再像我们之前那样完全参照数学模型精准建模，因为现实生活中的数据不一定完全与理论模型对等，就像我们之前谈过的**Black-Scholes**模型中出现的对数正态模型。

现逐步介绍如何构建基于 **PyTorch** 的量子生成对抗网络 (**qGAN**) 算法。

6.1 算法思路介绍

Black-Scholes 模型假设欧洲看涨期权的到期现货价格 S_T 是对数正态分布的。因此，我们可以在对数正态分布的样本上训练 **qGAN**，并将其结果用作期权的不确定性模型。

qGAN是一种用于生成建模任务的混合量子-经典算法。该算法利用量子生成器 G_θ (即参数化量子电路) 和经典判别器 D_ϕ (神经网络) 之间的相互作用，学习给定训练数据的底层概率分布。

生成器和判别器在交替优化步骤中进行训练，生成器旨在生成被判别器分类为训练数据值 (即来自真实训练分布的概率) 的概率，而判别器则试图区分原始分布和生成器生成的概率 (换句话说，区分真实分布和生成分布)。最终目标是让量子生成器学习目标概率分布的表示。

训练好的量子生成器可以用于加载一个量子态，该量子态是目标分布的近似模型。

用于加载随机分布的 qGANs

给定 k 维数据样本，我们使用量子生成对抗网络 (qGAN) 来学习一个随机分布，并将其直接加载到量子态中：

$$|g_\theta\rangle = \sum_{j=0}^{2^n-1} \sqrt{p_\theta^j} |j\rangle$$

其中 p_θ^j 描述了基态 $|j\rangle$ 的出现概率。

qGAN 训练的目标是生成一个状态 $|g_\theta\rangle$ ，其中 p_θ^j (对于 $j \in \{0, \dots, 2^n - 1\}$) 描述了一个接近训练数据 $X = \{x^0, \dots, x^{k-1}\}$ 底层分布的概率分布。

6.2 数据和表示

首先，我们需要加载我们的训练数据 X 。

在本教程中，训练数据由二维多变量正态分布给出。

生成器的目标是学习如何表示这种分布，训练好的生成器应对应于一个 n 量子比特的量子态

$$|g_{\text{trained}}\rangle = \sum_{j=0}^{k-1} \sqrt{p_j} |x_j\rangle,$$

其中基态 $|x_j\rangle$ 表示训练数据集中的数据项 $X = \{x_0, \dots, x_{k-1}\}$, $k \leq 2^n$, p_j 指的是 $|x_j\rangle$ 的采样概率。

如果我们使用 3 个量子比特来表示一个特征，我们有 $2^3 = 8$ 个离散值。

```
1 import torch
2 from qiskit_machine_learning.utils import algorithm_globals
```

我们固定维度数、离散化数量，并计算所需的量子比特数量为 $2^3 = 8$ 。

```
1 import numpy as np
2
3 num_dim = 2
4 num_discrete_values = 8
5 num_qubits = num_dim * int(np.log2(num_discrete_values))
```

使用对数正态分布生成了一维正态分布数据，并计算了相关的概率密度函数值。代码还设置了期权的参数，并计算了由对数正态分布得到的参数。

```

1 from scipy.stats import norm, lognorm
2
3 # 期权的参数(考虑的随机分布的参数)
4 S = 2.0 # 初始化股票价格
5 vol = 0.6 # 波动率
6 r = 0.05 # 无风险利率
7 T = 60 / 365 # 到期时间
8
9 # 由对数正态分布得到的参数
10 mu = (r - 0.5 * vol**2) * T + np.log(S)
11 sigma = vol * np.sqrt(T)
12 mean = np.exp(mu + sigma**2 / 2)
13 variance = (np.exp(sigma**2) - 1) * np.exp(2 * mu + sigma**2)
14 stddev = np.sqrt(variance)
15 # 为了计算方便, 我们将股票价格的最低值和最高值设置为均值的3倍标准差
16 low = np.maximum(0, mean - 3 * stddev)
17 high = mean + 3 * stddev
18
19
20 # 生成一维正态分布数据
21 coords = np.linspace(low, high, num_discrete_values)
22 rv = lognorm(loc=mu, scale=sigma, s=num_discrete_values)
23 grid_elements = np.transpose([np.tile(coords, 1)])
24 prob_data = rv.pdf(coords)
25 prob_data = prob_data / np.sum(prob_data)
26 print(grid_elements)
27

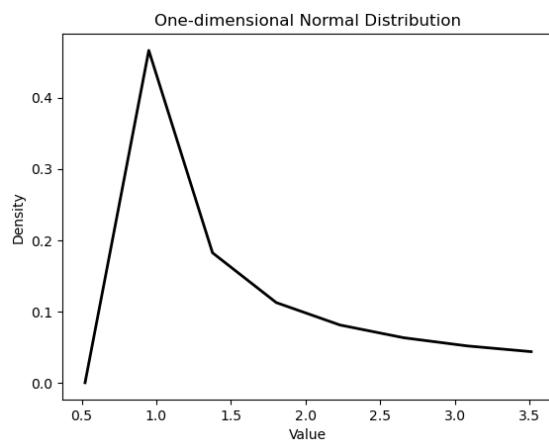
```

让我们可视化我们的分布。它是一个在离散网格上满足对数正态分布点的概率分布。

```

1 import matplotlib.pyplot as plt
2 from matplotlib import cm
3
4 # 绘制正态分布的概率密度函数
5 plt.plot(coords, prob_data, 'k', linewidth=2)
6 plt.title("One-dimensional Normal Distribution")
7 plt.xlabel("value")
8 plt.ylabel("Density")
9 plt.show()

```



6.3 神经网络的定义

在本节中，我们定义两个神经网络，如上所述：

- 量子生成器作为量子神经网络。
- 经典判别器作为基于 PyTorch 的神经网络。

6.3.1 量子神经网络 `ansatz` 的定义

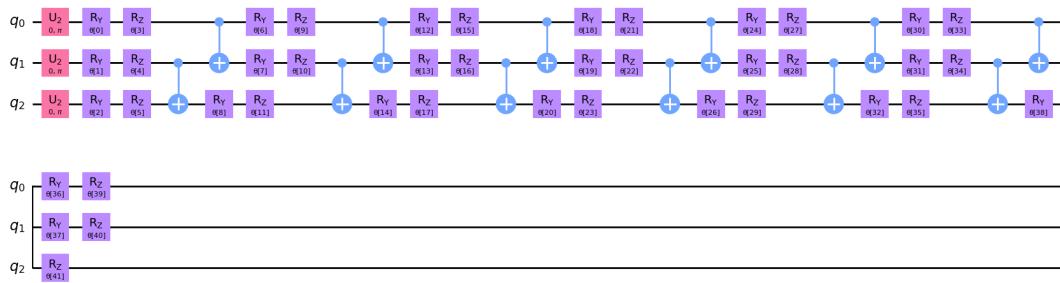
现在，我们定义参数化量子电路 $G(\theta)$ ，其中 $\theta = \{\theta_1, \dots, \theta_k\}$ ，它将用于我们的量子生成器。

为了实现量子生成器，我们选择一个硬件高效的 `ansatz`，重复 6 次。`ansatz` 实现了 R_Y 、 R_Z 旋转和 CX 门，以均匀分布作为输入状态。值得注意的是，对于 $k > 1$ ，生成器的参数必须仔细选择。例如，电路深度应大于 1，因为更高的电路深度能够表示更复杂的结构。在这里，我们构建了一个相当深的电路，具有大量参数，以便能够充分捕捉和表示分布。

```

1 | from qiskit import QuantumCircuit
2 | from qiskit.circuit.library import EfficientSU2
3 |
4 | qc = QuantumCircuit(num_qubits)
5 | qc.h(qc.qubits)
6 |
7 | ansatz = EfficientSU2(num_qubits, reps=6)
8 | qc.compose(ansatz, inplace=True)
9 |
10 | qc.decompose().draw(output="mpl", style="clifford")

```



打印可训练参数的数量。

```
1 | qc.num_parameters #42
```

6.3.2 量子生成器的定义

我们首先通过为 `ansatz` 创建一个采样器来定义生成器。参考实现是基于状态向量的实现，因此它返回电路执行结果的精确概率。在这种情况下，实现从测量的准概率分布中采样概率。

```

1 | from qiskit.primitives import StatevectorSampler as Sampler
2 |
3 | sampler = Sampler()

```

接下来，我们定义一个函数，从给定的参数化量子电路创建量子生成器。在这个函数内部，我们创建一个神经网络，该网络返回由底层采样器评估的准概率分布。我们固定 `initial_weights` 以确保结果的可重复性。最后，我们将创建的量子神经网络包装在 `TorchConnector` 中，以便使用基于 `PyTorch` 的训练。

```

1 | from qiskit_machine_learning.connectors import TorchConnector
2 | from qiskit_machine_learning.neural_networks import SamplerQNN

```

```
3
4 def create_generator() -> TorchConnector:
5     qnn = SamplerQNN(
6         circuit=qc,
7         sampler=sampler,
8         input_params=[],
9         weight_params=qc.parameters,
10        sparse=False,
11    )
12
13    initial_weights =
14        algorithm_globals.random.random(qc.num_parameters)
15
16    return TorchConnector(qnn, initial_weights)
```

6.3.3 经典判别器的定义

接下来，我们定义一个基于 **PyTorch** 的经典神经网络，表示经典判别器。底层梯度可以通过 **PyTorch** 自动计算。

```
1 from torch import nn
2
3 class Discriminator(nn.Module):
4     def __init__(self, input_size):
5         super(Discriminator, self).__init__()
6
7         self.linear_input = nn.Linear(input_size, 20)
8         self.leaky_relu = nn.LeakyReLU(0.2)
9         self.linear20 = nn.Linear(20, 1)
10        self.sigmoid = nn.Sigmoid()
11
12    def forward(self, input: torch.Tensor) -> torch.Tensor:
13        x = self.linear_input(input)
14        x = self.leaky_relu(x)
15        x = self.linear20(x)
16        x = self.sigmoid(x)
17
18    return x
```

6.3.4 创建生成器和判别器

现在我们创建一个生成器和一个判别器。

```
1 generator = create_generator()  
2 discriminator = Discriminator(num_dim)
```

6.4 设置训练循环

我们设置：

- 生成器和判别器的损失函数。
- 两者的优化器。
- 一个实用的绘图函数，用于可视化训练过程。

6.4.1 损失函数的定义

我们希望使用二元交叉熵作为损失函数来训练生成器和判别器：

$$L(\theta) = \sum_j p_j(\theta) [y_j \log(x_j) + (1 - y_j) \log(1 - x_j)],$$

其中 x_j 指的是数据样本， y_j 指的是相应的标签。

由于 PyTorch 的 [binary_cross_entropy](#) 对权重不可微分，我们手动实现损失函数，以便能够评估梯度。

```
1 def adversarial_loss(input, target, w):  
2     bce_loss = target * torch.log(input) + (1 - target) *  
3         torch.log(1 - input)  
4     weighted_loss = w * bce_loss  
5     total_loss = -torch.sum(weighted_loss)  
6     return total_loss
```

6.4.2 优化器的定义

为了训练生成器和判别器，我们需要定义优化方案。在接下来的步骤中，我们使用一种基于动量的优化器，称为 Adam，详见 [Adam: A Method for Stochastic Optimization](#)。

```
1 from torch.optim import Adam
2
3 lr = 0.01 # 学习率
4 b1 = 0.7 # 第一个动量参数
5 b2 = 0.999 # 第二个动量参数
6
7 generator_optimizer = Adam(generator.parameters(), lr=lr, betas=
8     (b1, b2), weight_decay=0.005)
9 discriminator_optimizer = Adam(
10     discriminator.parameters(), lr=lr, betas=(b1, b2),
11     weight_decay=0.005
12 )
13
```

6.4.3 训练过程的可视化

我们将通过绘制训练过程中生成器和判别器损失函数的演变，以及训练分布和目标分布之间相对熵的进展，来可视化训练过程。我们定义一个函数来绘制损失函数和相对熵。每当一个训练周期完成时，我们调用这个函数。

当收集到两个周期的训练数据时，训练过程的可视化开始。

```
1 from IPython.display import clear_output
2
3 def plot_training_progress():
4     # 如果没有足够的数据，我们不绘制
5     if len(generator_loss_values) < 2:
6         return
7
8     clear_output(wait=True)
9     fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 9))
10
11     # 生成器损失
12     ax1.set_title("Loss")
13     ax1.plot(generator_loss_values, label="generator loss",
14             color="royalblue")
15     ax1.plot(discriminator_loss_values, label="discriminator loss",
16             color="magenta")
17
18     ax1.legend(loc="best")
```

```
16     ax1.set_xlabel("Iteration")
17     ax1.set_ylabel("Loss")
18     ax1.grid()
19
20     # 相对熵
21     ax2.set_title("Relative entropy")
22     ax2.plot(entropy_values)
23     ax2.set_xlabel("Iteration")
24     ax2.set_ylabel("Relative entropy")
25     ax2.grid()
26
27 plt.show()
```

6.5 模型训练

在训练循环中，我们不仅监控损失函数，还监控相对熵。相对熵描述了分布的距离度量。因此，我们可以用它来评估训练分布与目标分布的接近程度。

现在，我们准备训练我们的模型。训练模型可能需要一些时间，请耐心等待。

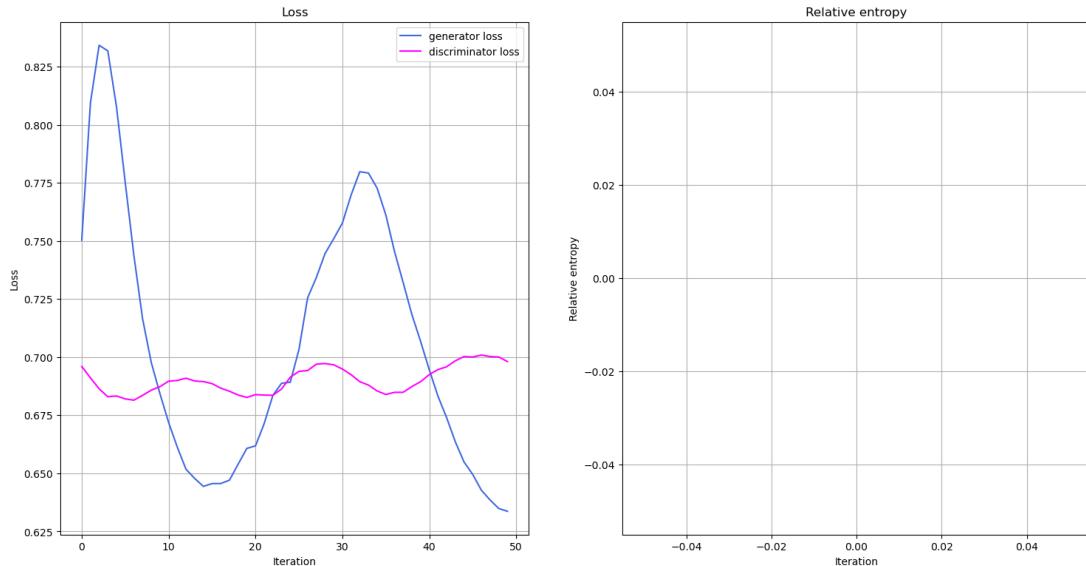
```
1 import time
2 from scipy.stats import multivariate_normal, entropy
3
4 n_epochs = 50
5
6 num_qnn_outputs = num_discrete_values**num_dim
7
8 generator_loss_values = []
9 discriminator_loss_values = []
10 entropy_values = []
11
12 start = time.time()
13 for epoch in range(n_epochs):
14
15     valid = torch.ones(num_qnn_outputs, 1, dtype=torch.float)
16     fake = torch.zeros(num_qnn_outputs, 1, dtype=torch.float)
17
18     # 配置输入
```

```
19     real_dist = torch.tensor(prob_data,
20                               dtype=torch.float).reshape(-1, 1)
21
22     # 配置样本
23     samples = torch.tensor(grid_elements, dtype=torch.float)
24     disc_value = discriminator(samples)
25
26     # 生成数据
27     gen_dist = generator(torch.tensor([])).reshape(-1, 1)
28
29     # 训练生成器
30     generator_optimizer.zero_grad()
31     generator_loss = adversarial_loss(disc_value, valid, gen_dist)
32
33     # 存储以便绘图
34     generator_loss_values.append(generator_loss.detach().item())
35
36     generator_loss.backward(retain_graph=True)
37     generator_optimizer.step()
38
39     # 训练判别器
40     discriminator_optimizer.zero_grad()
41
42     real_loss = adversarial_loss(disc_value, valid, real_dist)
43     fake_loss = adversarial_loss(disc_value, fake,
44                                  gen_dist.detach())
45     discriminator_loss = (real_loss + fake_loss) / 2
46
47     # 存储以便绘图
48
49     discriminator_loss_values.append(discriminator_loss.detach().item())
50
```

```

51     entropy_value = entropy(gen_dist.detach().squeeze().numpy(),
52                               prob_data)
53
54     entropy_values.append(entropy_value)
55
56 elapsed = time.time() - start
57 print(f"Fit in {elapsed:.2f} sec")

```



熵的结果不一定显示

6.6 显示训练结果

我们将训练分布的结果生成

首先，我们在关闭 PyTorch 自动梯度的情况下生成一个新的概率分布，因为我们不再训练模型。

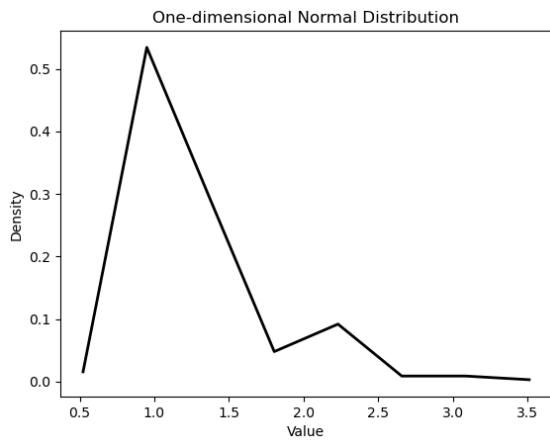
```

1 with torch.no_grad():
2     generated_probabilities = generator().numpy()

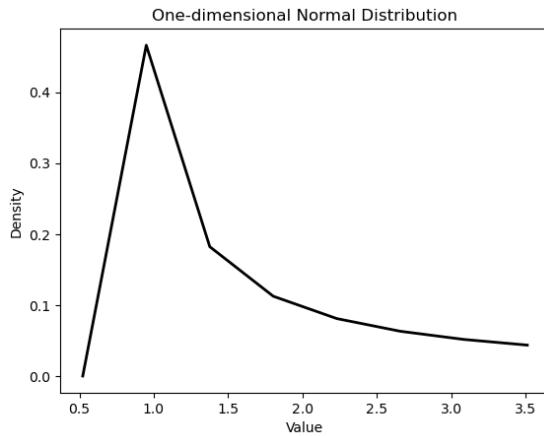
```

然后，我们绘制生成训练分布的结果

```
1 # 绘制正态分布的概率密度函数
2 plt.plot(coords, generated_probabilities, 'k', linewidth=2)
3 plt.title("One-dimensional Normal Distribution")
4 plt.xlabel("value")
5 plt.ylabel("Density")
6 plt.show()
7 print(generated_probabilities)
```



对比我们之前的



可以看出训练结果较好

6.7 欧式看涨期权定价

操作流程如本文核心所讲，只是对数正态分布模型换为了我们自定义的类

这里的操作就是我们整篇文章所描述的方法，其实思路与上文一致，这里不再赘述，在这里我们继承了**QuantumCircuit**类，实现了将量子对抗网络生成的数据导入到不确定性模型，其实现思路如下：

```

1  class MyLogNormalDistribution(QuantumCircuit):
2
3      def __init__(
4          self,
5          num_qubits: Union[int, List[int]],
6          gene_prpo: List[float],
7          bounds: Optional[Union[Tuple[float, float],
8              List[Tuple[float, float]]]] = None,
9          name: str = "P(X)",
10         ) -> None:
11             inner = QuantumCircuit(num_qubits, name=name)
12
13             x = np.linspace(bounds[0], bounds[1], num=2**num_qubits)
14
15             # compute the normalized, truncated probabilities
16             normalized_probabilities = gene_prpo / np.sum(gene_prpo)
17
18             # store as properties
19             self._values = x
20             self._probabilities = gene_prpo
21             self._bounds = bounds
22
23             super().__init__(*inner.qregs, name=name)
24
25             init_params = np.sqrt(gene_prpo).astype(float)
26             init_params = init_params / np.linalg.norm(init_params)
27             initialize = Initialize(init_params)
28             circuit = initialize.gates_to_uncompute().inverse()
29             inner.compose(circuit, inplace=True)
30             self.append(inner.to_gate(), inner.qubits)
31
32             @property
33             def values(self) -> np.ndarray:
34                 """Return the discretized points of the random variable."""
35                 return self._values

```

```

36     def probabilities(self) -> np.ndarray:
37         """Return the sampling probabilities for the values."""
38         return self._probabilities
39
40     @property
41     def bounds(self) -> Union[Tuple[float, float],
42                               List[Tuple[float, float]]]:
43         """Return the bounds of the probability distribution."""
44         return self._bounds

```

`MyLogNormalDistribution` 是一个继承自 `QuantumCircuit` 的类，用于在量子电路中表示对数正态分布。其构造函数 `__init__` 接受四个参数：

- `num_qubits` 指定量子比特的数量或列表
- `gene_prpo` 是生成的概率列表
- `bounds` 是可选的边界值
- `name` 是电路的名称

首先，创建一个名为 `inner` 的 `QuantumCircuit` 对象，并使用 `np.linspace` 根据边界值生成离散化的点。

接下来，计算并归一化生成的概率 `gene_prpo`，并将这些值存储为类的属性 `_values`、`_probabilities` 和 `_bounds`。

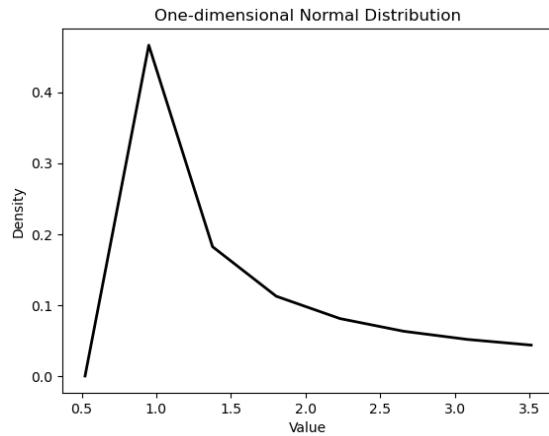
然后，通过调用父类 `QuantumCircuit` 的构造函数来初始化电路。为了初始化量子态，计算 `gene_prpo` 的平方根并归一化，生成初始化参数 `init_params`。使用这些参数创建 `Initialize` 对象，并生成逆初始化电路 `circuit`，将其合成到 `inner` 电路中。最后，将 `inner` 电路转换为门并附加到当前电路中。

此外，`MyLogNormalDistribution` 类还定义了三个属性方法：

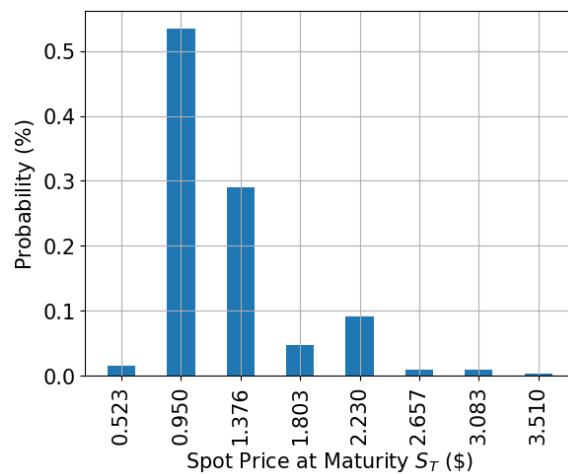
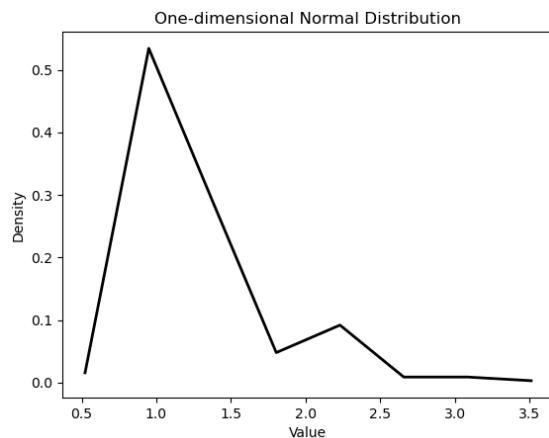
- `values` 返回离散化的随机变量点
- `probabilities` 返回这些点的采样概率
- `bounds` 返回概率分布的边界。这些属性方法提供了对类内部存储数据的访问接口

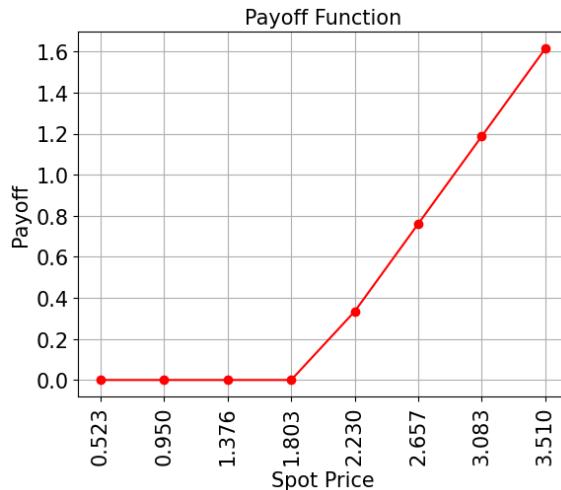
6.8 测试结果

6.8.1 实际数据



6.8.2 实际训练数据





- 精准的期望收益和Delta

```

生成 + 代码 + Markdown

# 计算精确的期望值(归一化到[0,1])和Delta值
exact_value = np.dot(mylog.probabilities, y)
exact_delta = sum(mylog.probabilities[x >= strike_price])
print("exact expected value:\t%.4f" % exact_value)
print("exact delta value: \t%.4f" % exact_delta)
26] ✓ 0.0s
.. exact expected value: 0.0525
exact delta value: 0.1123

```

- 期望收益

```

conf_int = np.array(result.confidence_interval_processed)
print("Exact value: \t%.4f" % exact_value)
print("Estimated value: \t%.4f" % (result.estimation_
print("Confidence interval:\t%.4f, %.4f" % tuple(conf_i
20] ✓ 0.0s
.. Exact value: 0.0525
Estimated value: 0.0738
Confidence interval: [0.0614, 0.0861]

```

- Delta

```

conf_int = np.array(result_delta.confidence_interval_
print("Exact delta: \t%.4f" % exact_delta)
print("Estimated value: \t%.4f" % european_call_de
print("Confidence interval: \t%.4f, %.4f" % tuple(c
1] ✓ 0.0s
.. Exact delta: 0.1123
Estimated value: 0.1126
Confidence interval: [0.1111, 0.1141]

```

6.9 结果说明**

由于训练参数并不是很好，我们得到的期望收益和**Delta**均不是很精准。

调参是世界上最枯燥最无聊的工作，我们应该把有限的时间花在有意义的事情上面。笔者这里不是很想继续训练下去了，如果想要训练使之性能达到最优的，可以自己尝试一下，选做**Bonus**就做到这里

7 环境配置

给出的例程主要基于**Qiskit**实现，这里以在**VSCode**中配置**Python**编程环境为例。

建议使用**Python**的版本为**3.10**。

```
1 | conda create -n "summer-camp-qf-2023" python=3.10
2 | conda activate summer-camp-qf-2023
```

安装**Qiskit Finance**。

```
1 | pip install qiskit-finance
```

安装必要的画图工具。

```
1 | pip install matplotlib
2 | pip install py latexenc
```

安装必要的优化算法包。

```
1 | pip install scikit-opt
```

安装可用于**Jupyter**的**IPython**内核，即可在**VSCode**中运行**Jupyter Notebook**。

```
1 | pip install ipykernel
```

撰写选做任务需要安装pytorch**，安装包较大，请读者量力而行(x