QISKIT FINANCE

UNIVERSIDAD
COMPLUTENSE
MADRID

Proyecto final Arquitectura y Programación
de Computadores Cuánticos

Alberto Bañegil Díaz

Facultad de Informática, Universidad Complutense de Madrid
Mayo 2021

ÍNDICE:

INTRODUCCIÓN:

Vamos a abordar de modo muy práctico, las librerías e implementaciones que tiene Qiskit para las finanzas, la motivación de este proyecto viene dada por la velocidad de procesamiento de datos que nos aporta la computación cuántica, basándose en su capacidad de paralelización de procesamiento.
A día de hoy necesitamos no solo procesar grandes volúmenes de datos, sino procesar datos en tiempo real, a gran velocidad. En concreto, en finanzas hay un pequeño nicho de mercado llamado 'High Frecuency Trading', en el que llegar 1 nanosegundo antes que el otro, abre la barrera entre ganadores y perdedores. Dado que la computación clásica ha llegado a un punto de estancamiento, gracias a la computación cuántica y los algoritmos cuánticos, sería posible seguir procesando datos a mayor velocidad, esto se traduce desde ejecutar pruebas o Backtesting de optimización que en computación clásica tardarían considerablemente más tiempo, hasta el ejemplo mencionado anteriormente en el que la diferencia entre 1 milisegundo y 5 milisegundos es una eternidad.
Sin ser este un tutorial de cómo hacer un análisis exhaustivo de datos, sino más bien exponer los avances en Qiskit y en computación cuántica que se están aplicando a día de hoy, y sobre todo, plantearlo de una manera que pueda ser escalable.
En los 2 últimos apartados, compararemos las implementaciones con implementaciones clásicas en Python y C++, ya que son los lenguajes más utilizados en este sector.

1.- Carga y procesamiento de datos de series temporales del mercado de valores:

En este apartado vamos a ver cómo generar series temporales pseudoaleatorias y cómo descargar series temporales reales de las principales bolsas de valores del mundo. Un generador de números aleatorio cuántico sería posible, en este caso, Qiskit implementa uno clásico (pseudoaleatorio), que además, al tratarse de series temporales, genera valores dentro de unos parámetros, para que se acerquen lo más posible a la realidad.

[1]:Comenzamos importando las librerías pertinentes. Como podemos observar, Qiskit ya ofrece librerías de proveedores de datos muy útiles como Yahoo, Wikipedia o Quandl por ejemplo.
A partir de ahora llamaremos “Ticker” a una serie temporal de precios, en nuestro caso, precios de activos como el oro o una compañía.

```
%matplotlib inline

from qiskit_finance import QiskitFinanceError

from qiskit_finance.data_providers import *

import datetime

import matplotlib.pyplot as plt

from pandas.plotting import register_matplotlib_converters

register_matplotlib_converters()
```

[2]:Qiskit ofrece una Clase llamada ‘RandomDataProvider’: que provee datos de series temporales pseudoaletorios, y cuenta con funciones que devuelven un vector que contiene el valor medio de cada activo o devuelve la matriz de covarianza, entre otras. Añadimos dos series temporales pseudoaletorias de ejemplo ["TICKER1", "TICKER2"] e inicializamos las fechas de inicio y fin de ambas..

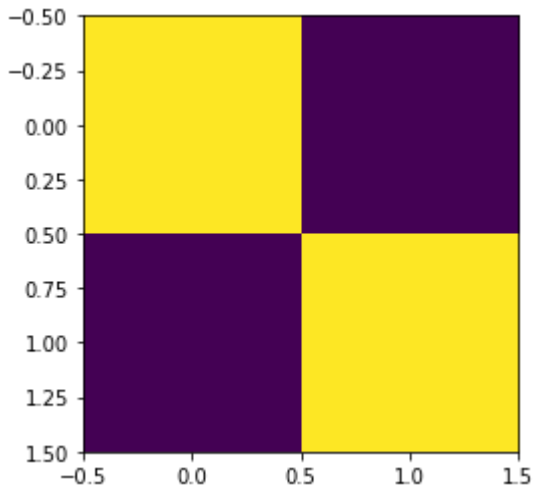
```
data = RandomDataProvider(tickers=["TICKER1", "TICKER2"],

                start = datetime.datetime(2016, 1, 1),

                end = datetime.datetime(2016, 1, 30),

                seed = 1)

data.run()
```

[3]:Llamamos a funciones de la clase que hacen los siguiente: obtienen un vector con el valor medio de cada activo, se comprueba su similitud entre ambas series temporales y a continuación se aplica la covarianza de ambas.

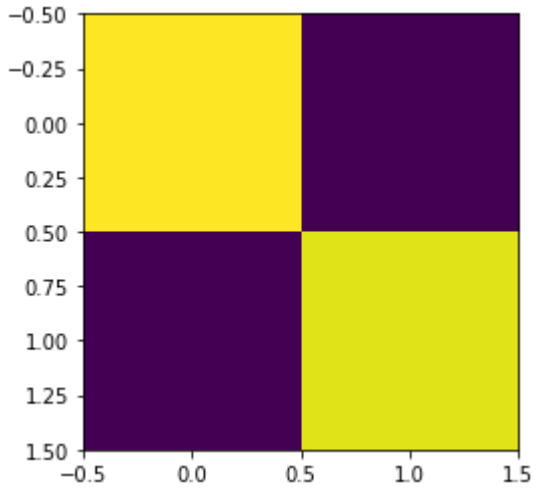
```
means = data.get_mean_vector()

print("Media:")

print(means)

rho = data.get_similarity_matrix()

print("Medición de similitud entre series temporales:")

print(rho)

plt.imshow(rho)

plt.show()

cov = data.get_covariance_matrix()

print("Matriz de covarianza:")

print(cov)

plt.imshow(cov)

plt.show()
```

Media:
[33.97683271 97.61130683]

Medición de similitud entre series temporales:
[[1.00000000e+00 5.41888011e-04]
[5.41888011e-04 1.00000000e+00]]



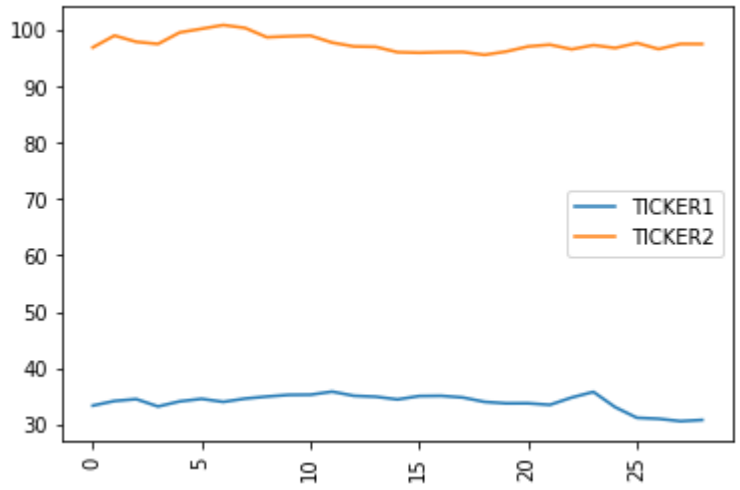
Matriz de covarianza:
[[2.08413157 0.20842107]
[0.20842107 1.99542187]]



[4]:Como hemos observado, el valor medio de TICKER1 es 33.97683271 y el de TICKER2 es 97.61130683, distan bastante en precio, en la matriz de similitud muestra que guarda cierta relación, por el hecho de ser series temporales que no fluctúan más allá del 10% en sus precios, pero en la matriz de covarianza, aunque haría falta hacer un estudio de la matriz de correlación, muestra que no están relacionadas. En el siguiente apartado comprobamos un ejemplo totalmente distinto.

```
print("La evolución subyacente de los precios de las acciones:")

for (cnt, s) in enumerate(data._tickers):

   plt.plot(data._data[cnt], label=s)

plt.legend()

plt.xticks(rotation=90)

plt.show()

for (cnt, s) in enumerate(data._tickers):

   print(s)

   print(data._data[cnt])
```

La evolución subyacente de los precios de las acciones:



[5]: Aquí empezamos a comprobar que el generador de precios pseudoaleatorio genera rangos que pueden ser muy dispares entre diferentes compañías, también cabe notar que en intervalo de tiempo en este caso es más grande (1 año y 1 mes).

```
data = RandomDataProvider(tickers=["CompanyA", "CompanyB", "CompanyC"],

                start = datetime.datetime(2015, 1, 1),

                end = datetime.datetime(2016, 1, 30),

                seed = 1)

data.run()

for (cnt, s) in enumerate(data._tickers):

   plt.plot(data._data[cnt], label=s)

plt.legend()

plt.xticks(rotation=90)

plt.show()
```

Acceso a series temporales de precios de cierre

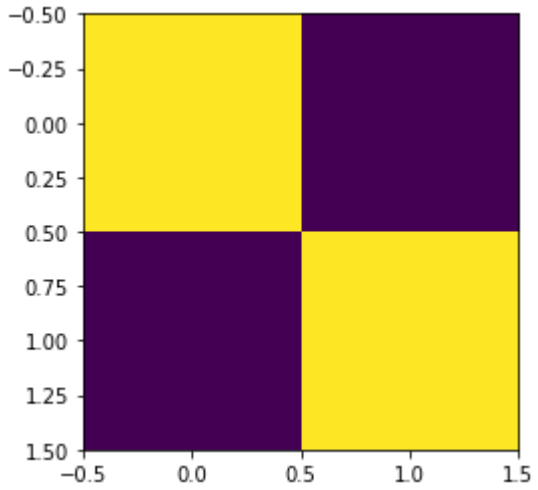
La obtención de precios en tiempo real son de pago, con Quandl podemos obtener datos históricos en un amplio rango de intervalos de tiempo, desde años hasta ticks, un tick no es un intervalo de tiempo, pero como se ha llegado a un punto en el que en 1 milisegundo puede haber varias transacciones, cada transacción hace referencia a un tick. En concreto, obtener datos de cierre a nivel de tick también suele ser de pago, pero con Quandl es gratis https://www.quandl.com/?modal=register. Vamos a obtener los datos básicos de 2 bancos estadounidenses, JPMorgan y Morgan Stanley, y aplicaremos las mismas funciones de similitud y covarianza que hemos hecho en el primer ejemplo con las series temporales pseudoaleatorias. Con las clase 'Wikipedia DataProvider' accedemos de la misma manera que con la clase random, pero en este caso accedemos a datos reales.

```
import quandl

stocks = ["JPM", "MS"]

token = "2rnPhfQavjPBb_xyGTH1"

if 1:

   try:

       wiki = WikipediaDataProvider(

                          token = token,

                          tickers = stocks,

                          start = datetime.datetime(2004,1,1),

                          end = datetime.datetime(2016,1,30))
```

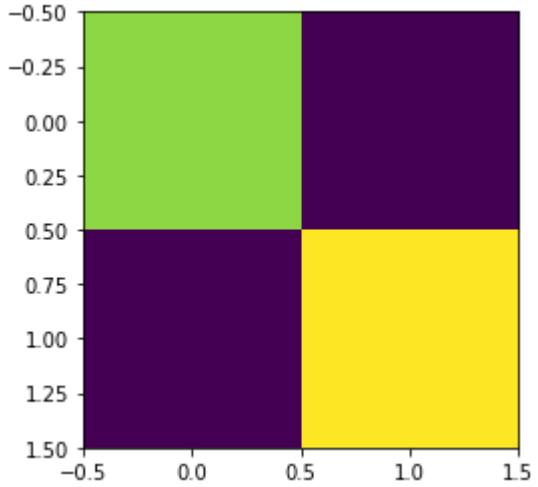
```
        wiki.run()
    except QiskitFinanceError as ex:
        print(ex)
        print("Error extrayendo datos.")
```

```
if 1:
    if wiki._data:
        if wiki._n <= 1:
            print("No hay suficientes datos de wiki para trazar covarianza o
                  semejanza de series temporales. Utilice al menos dos tickers.")
        else:
            rho = wiki.get_similarity_matrix()
            print("Medición de similitud entre series temporales:")
            print(rho)
            plt.imshow(rho)
            plt.show()
            cov = wiki.get_covariance_matrix()
            print("Matriz de covarianza:")
            print(cov)
            plt.imshow(cov)
            plt.show()
    else:
        print('No hay wiki datos cargados.')
```

Medición de similitud entre series temporales:
[[1.00000000e+00 4.09281925e-05] [4.09281925e-05 1.00000000e+00]]



Matriz de covarianza::[[115.10692101 9.47724962] [9.47724962 136.90714619]]

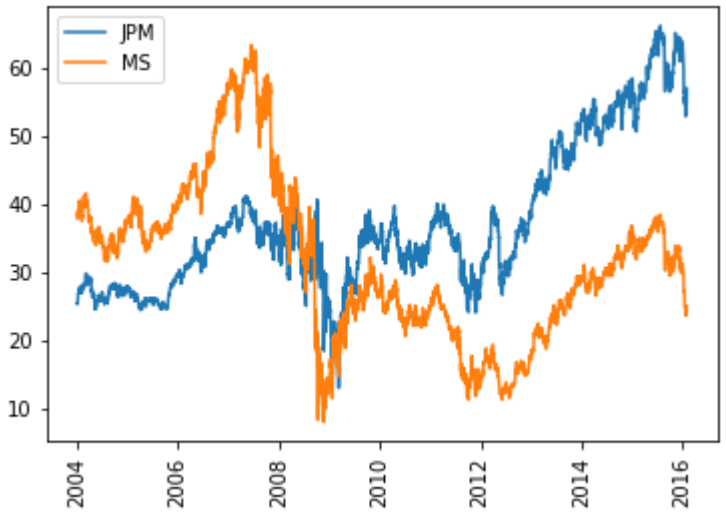


[6]: Observamos que en este caso la matriz de similitud indica que guarda cierta relación, a la par que en la matriz de covarianza. Sin ser esto un tutorial exhaustivo de cómo hacer correctamente un análisis estadístico, sino más bien exponer cómo podemos extraer datos para su posterior análisis en un entorno cuántico y mostrar algunas funciones estadísticas básicas. En los siguientes temas entraremos más en profundidad en cómo podemos abordar un análisis de series temporales con algoritmos cuánticos.

```
if 1:

   if wiki._data:

       print("The underlying evolution of stock prices:")

       for (cnt, s) in enumerate(stocks):

           plt.plot(wiki._data[cnt], label=s)
```

```
        plt.legend()

        plt.xticks(rotation=90)

        plt.show()



        for (cnt, s) in enumerate(stocks):

            print(s)

            print(wiki._data[cnt])

    else:

        print('La evolución subyacente de los precios de las acciones.')
```

La evolución subyacente de los precios de las acciones:



Configurar token para acceder a series temporales recientes y detalladas. En general, de pago.

Para descargar datos profesionales que usan los fondos de inversión, debes tener una licencia NASDAQ. De esta manera podemos acceder a datos en tiempo real y recientes, dado que, cuanto más reciente es el dato, más caro es. Si no tienes la licencia NASDAQ Data on Demand, puedes comunicarte con NASDAQ https://business.nasdaq.com/intel/GIS/Nasdaq-Data-on-Demand.html para obtener una licencia de prueba o de pago. El constructor 'Data On Demand Provider' tiene un argumento opcional 'verify', que puede ser 'None', una cadena o un booleano. Si es

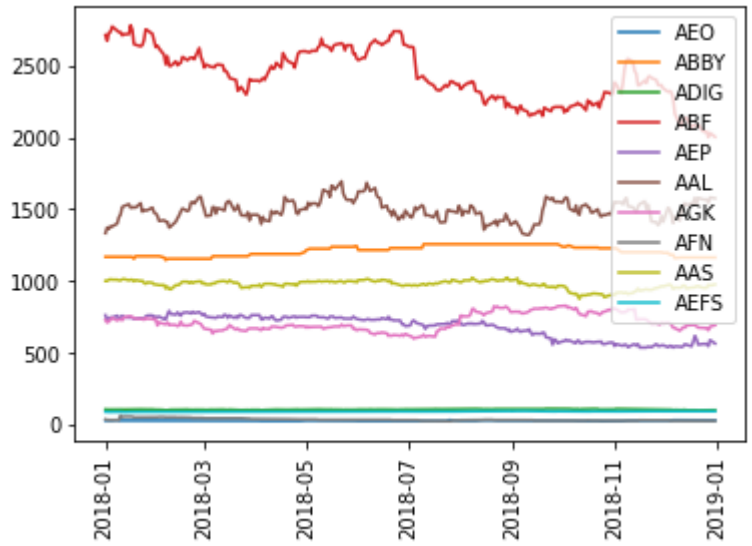
‘None’, se utilizarán certificados(predeterminado). Si verify es una cadena, debería apuntar a un certificado para la conexión HTTPS a NASDAQ (dataondemand.nasdaq.com), ya sea en forma de archivo CA_BUNDLE o un directorio donde buscar.

```
if 1:

   try:

       nasdaq = DataOnDemandProvider(token = token,

                    tickers = ["GOOG", "AAPL"],

                    start = datetime.datetime(2016,1,1),

                    end = datetime.datetime(2016,1,2))

       nasdaq.run()

       for (cnt, s) in enumerate(nasdaq._tickers):

           plt.plot(nasdaq._data[cnt], label=s)

       plt.legend()

       plt.xticks(rotation=90)

       plt.show()

   except QiskitFinanceError as ex:

       print(ex)

       print("Error retrieving data.")
```

Otro proveedor importante de datos bursátiles es Exchange Data International (EDI), cuya API se puede utilizar para consultar más de 100 mercados emergentes y fronterizos que son África, Asia, Lejano Oriente, América Latina y Medio Oriente. https://www.exchange-data.com/pricing-data/adjusted-prices.php#exchange-coverage para obtener una descripción general de la cobertura.

```
if 1:

   try:

       lse = ExchangeDataProvider(token = token,

                    tickers = ["AEO", "ABBY", "ADIG", "ABF",
```

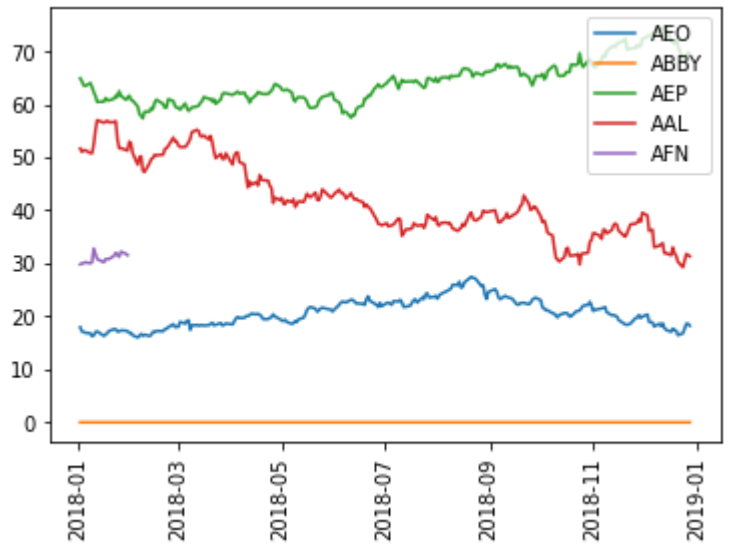
```
                                  "AEP", "AAL", "AGK", "AFN", "AAS", "AEFS"],
                    stockmarket = StockMarket.LONDON,
                    start=datetime.datetime(2018, 1, 1),
                    end=datetime.datetime(2018, 12, 31))
        lse.run()
        for (cnt, s) in enumerate(lse._tickers):
            plt.plot(lse._data[cnt], label=s)
        plt.legend()
        plt.xticks(rotation=90)
        plt.show()
    except QiskitFinanceError as ex:
        print(ex)
        print("Error retrieving data.")
```



También se puede acceder a Yahoo Finance Data, sin necesidad de token de Quandl ni ningún otro requisito, desde Yahoo! Finanzas.

```
try:
    data = YahooDataProvider(
                tickers = ["AEO", "ABBY", "AEP", "AAL", "AFN"],
```

```
                start=datetime.datetime(2018, 1, 1),
                end=datetime.datetime(2018, 12, 31))
    data.run()
    for (cnt, s) in enumerate(data._tickers):
        plt.plot(data._data[cnt], label=s)
    plt.legend()
    plt.xticks(rotation=90)
    plt.show()
except QiskitFinanceError as ex:
    data = None
    print(ex)
```



2.- Pricing de activos de renta fija:

A continuación vamos a valorar un activo de renta fija, utilizando una distribución normal aleatoria multivariante, la ecuación básica que se utiliza en este tipo de activos es la mostrada abajo, que muestra una serie temporal en la que juegan un importante papel 2 variables: **ct**, que hace referencia al cash flow del activo, siendo **t** la fecha en la que ocurre, y **rt**, que hace referencia a la tasa de interés asociada al bono. **V** es el valor total del activo. Vamos a centrarnos en la variable **rt**, que se va a basar en un modelo de distribución cuántico de incertidumbre, también indicaremos parámetros como el número de qubits que utilizamos para representar las diferentes dimensiones del modelo de incertidumbre, y dos parámetros **[low,high]** que acota el rango del valor final estimado.

$$V = \sum_{t=1}^{T} \frac{c_t}{(1+r_t)^t}$$

[1]: importamos las librerías pertinentes, en concreto, entender la clase 'NormalDistribution' es esencial para comprender bien el siguiente apartado.

```
import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np

from qiskit import Aer, QuantumCircuit

from qiskit.utils import QuantumInstance

from qiskit.algorithms import IterativeAmplitudeEstimation, EstimationProblem

from qiskit_finance.circuit.library import NormalDistribution

backend = Aer.get_backend('statevector_simulator')
```

Modelo de incertidumbre

Dado $\otimes_{i=1}^{d}[low_i, high_i]$, vamos a volcar la distribución en un estado cuántico entre los límites **[low,high]** de 1 a **d** dimensiones, en nuestro ejemplo, la dimensión será de 2. Como mencionamos anteriormente, escogemos el número de qubits, que en nuestro ejemplo será de 2, esto significa que usaremos 4 puntos, ya que es 2 elevado a $\mathbf{n_i}$, donde i denota la dimensión.

$$|0\rangle_{n_1} \dots |0\rangle_{n_d} \mapsto |\psi\rangle = \sum_{i_1=0}^{2^n-1} \dots \sum_{i_d=0}^{2^n-1} \sqrt{p_{i_1,\dots,i_d}} |i_1\rangle_{n_1} \dots |i_d\rangle_{n_d},$$

$$\{0, \dots, 2^{n_j} - 1\} \ni i_j \mapsto \frac{high_j - low_j}{2^{n_j} - 1} * i_j + low_j \in [low_j, high_j].$$

Pi….id denota las probabilidades correspondientes a la distribución volcada y discretizada
ij se asigna al mapa afín entre **[low,high]**.

[2]: especificamos el número de qubits que se utilizan para representar las diferentes dimensiones del modelo de incertidumbre, especificamos los límites [low,high] para las diferentes dimensiones. Aquí es donde tenemos que tener claro que parámetros estamos pasando a la clase NormalDistribution, dónde Σ sigma. Para especificar una distribución normal multivariante, num_qubits hay una lista de enteros, cada uno especificando cuántos qubits se utilizan para discretizar la dimensión respectiva. Los argumentos mu y sigma en este caso son un vector y una matriz cuadrada.

```
A = np.eye(2)
b = np.zeros(2)

num qubits = [2, 2]

low = [0, 0]

high = [0.12, 0.24]

mu = [0.12, 0.24]

sigma = 0.01*np.eye(2)

bounds = list(zip(low, high))

u = NormalDistribution(num_qubits, mu, sigma, bounds)
```

[3]: Aquí es donde está una de las claves de este tema, la función de la distribución normal se aplica de manera totalmente distinta en modo cuántico a modo clásico, está variable **u** al usaremos posteriormente para hacer el valor estimado del bono. Todas estas implementaciones usan funciones internas, en concreto, la implementación cuántica, que es la que nos interesa, usa puertas RY que rota sobre el eje Y, con una amplitud de 2 elevado a n. A continuación vamos a ver dos implementaciones clásicas de la distribución normal sencillas y triviales, ya que C++ y Python también tienen sus porpias librerías que implementan dicha función.

Código Python clásico:

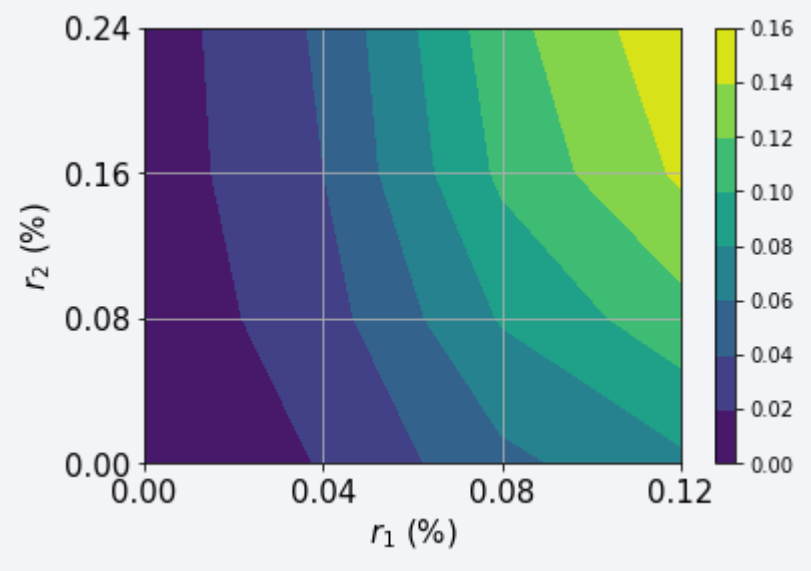
```
media = (1, 2)

sigma = [[0, 0], [0, 1]]

u = np.random.multivariate_normal (media, sigma, (3, 3))
```

Código C++ clásico:

```
#include <iostream>

#include <iomanip>

#include <string>

#include <map>

#include <random>

#include <cmath>

int main ()

{

     std :: random_device rd {};

     std :: mt19937 gen {rd ()};

     // los valores cercanos a la media son los más probables

     // La desviación estándar afecta la dispersión de los valores generados a
partir de la media..

     std :: normal_distribution <> d {5,2};

     std :: map <int, int> hist {};

     for (int n = 0; n <10000; ++ n) {

         ++ hist [std :: round (d (gen))];

     }

     for (auto p: hist) {

         std :: cout << std :: setw (2)

                     << p.first << '' << std :: string (p.second / 200, '*');}}
```

[4]: ploteamos el contorno de la función de densidad de probabilidad cuántica

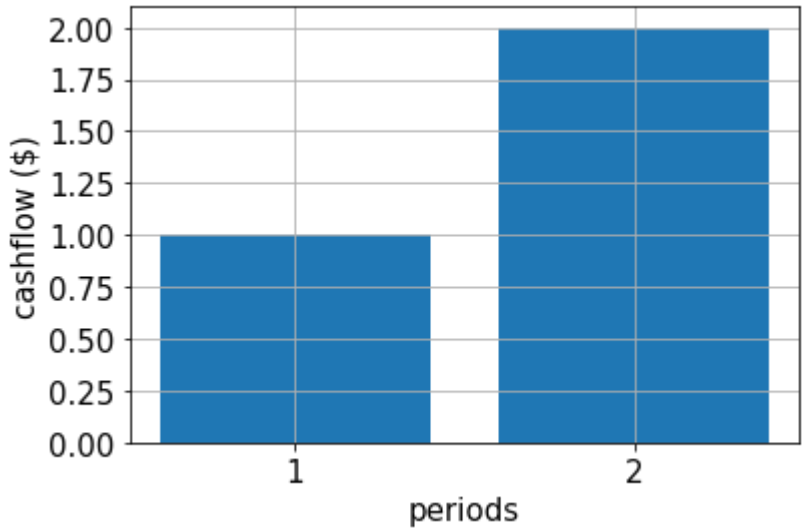
```
x = np.linspace(low[0], high[0], 2**num_qubits[0])

y = np.linspace(low[1], high[1], 2**num_qubits[1])

z = u.probabilities.reshape(2**num_qubits[0], 2**num_qubits[1])

plt.contourf(x, y, z)

plt.xticks(x, size=15)

plt.yticks(y, size=15)

plt.grid()

plt.xlabel('$r_1$ (%)', size=15)

plt.ylabel('$r_2$ (%)', size=15)

plt.colorbar()

plt.show()
```



Flujo de caja, función de pago y valor esperado exacto

A continuación, definimos el cash flow por período, la función de pago resultante y evaluamos el valor esperado exacto. Para la función de pago, primero usamos una aproximación de primer orden

```
cf = [1.0, 2.0]

periods = range(1, len(cf) + 1)

plt.bar(periods, cf)

plt.xticks(periods, size=15)

plt.yticks(size=15)

plt.grid()

plt.xlabel('periods', size=15)

plt.ylabel('cashflow ($)', size=15)

plt.show()
```



[5]: estimamos el valor real y evaluamos la aproximación lineal del valor real de las tasas de interés. Este código hace referencia a la ecuación mostrada al principio del cálculo de V, pero desde un punto de vista cuántico, con las dimensiones y qubits declarados anteriormente. También se adjunta el código en python clásico, para que se vea la diferencia con el cuántico.

```
cnt = 0

exact_value = 0.0

for x1 in np.linspace(low[0], high[0], pow(2, num_qubits[0])):

    for x2 in np.linspace(low[1], high[1], pow(2, num_qubits[1])):

        prob = u.probabilities[cnt]
```

```
        for t in range(len(cf)):

            exact_value += prob * (cf[t]/pow(1 + b[t], t+1) -
(t+1)*cf[t]*np.dot(A[:, t], np.asarray([x1, x2]))/pow(1 + b[t], t+2))

        cnt += 1

print('Exact value:    \t%.4f' % exact_value)
```

Valor exacto: 2.1942

[6]: Como podemos observar, se parece mucho, la diferencia, y por lo que este código parece más simple, pero es menos completo, es por el parámetro **r**, en este caso introducimos 0.056 manualmente, en el ejemplo cuántico, creamos un espectro de posibilidades en el que hay **d** dimensiones y **q** qubits, en nuestro caso, 2 y 2.

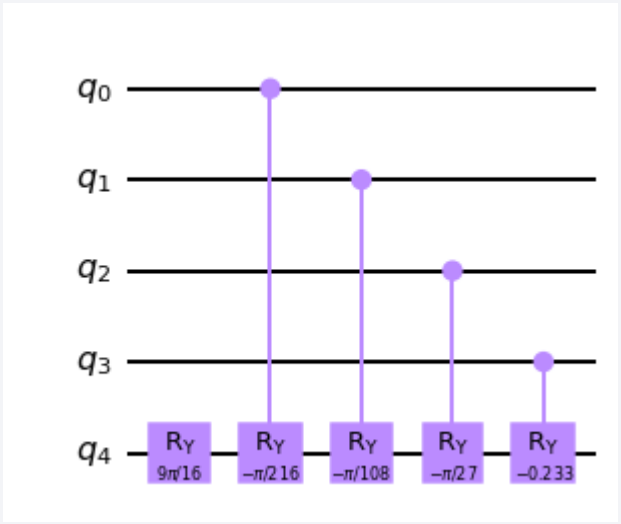
Código Python clásico:

```
def bonds_price_discrete (times, cashflows, r):

    p = 0

      for i in range (len (times)):

        p + = cashflows [i] / np.power ((1 + r), times [i])

    return p

import numpy as np



c = np.array ([1, 2])

t = np.arange (1, 2)

r = 0.056

d = (1. / np.power ((1 + r), t))

B = np.sum (d * c)

print ('Exact value = {: .3f}'. format (bonds_price_discrete (t, c, r)))
```

[7]: Especificamos valor de aproximación y obtenemos la appFactory de renta fija con la clase 'FixedIncomePricing'. Además, vamos a mostrar nuestro primer circuito cuántico, que

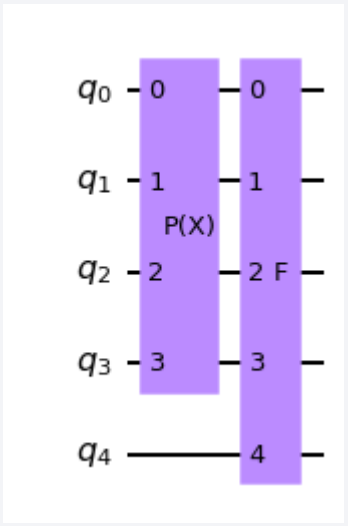
de momento, es una plantilla. Todavía no hemos cargado la distribución multivariante cuántica calculada anteriormente.

```
c_approx = 0.125

from qiskit_finance.applications.estimation import FixedIncomePricing

fixed_income = FixedIncomePricing(num_qubits=num_qubits, pca_matrix=A,
initial_interests=b,
                                  cash_flow=cf, rescaling_factor=c_approx,
bounds=bounds,
                                  uncertainty_model=u)

fixed_income._objective.draw()
```



[8]: Cargamos la función de probabilidad y aplicamos la función

```
fixed_income_circ = QuantumCircuit(fixed_income._objective.num_qubits)

fixed_income_circ.append(u, range(u.num_qubits))

fixed_income_circ.append(fixed_income._objective,
range(fixed_income._objective.num_qubits))

fixed_income_circ.draw()
```



[9]: Establecemos la precisión del objetivo y el nivel de confianza (epsilon), construimos la estimación de la amplitud (alpha). Ahora solo queda evaluar el circuito, en este caso con 100 muestras y no en un simulador real, ya que se necesita un token de IBM Quantum experience que no se puede exponer públicamente, aunque los resultados son muy parecidos. Con la clase 'IterativeAmplitudeEstimation', La salida del algoritmo es una estimación que, con al menos probabilidad 1−α, difiere por épsilon del valor objetivo, donde se pueden especificar tanto alfa como épsilon, aplica iterativamente iteraciones de Grover cuidadosamente seleccionadas para encontrar una estimación de la amplitud objetivo.

```
epsilon = 0.01

alpha = 0.05

qi = QuantumInstance(Aer.get_backend('qasm_simulator'), shots=100)

problem = fixed_income.to_estimation_problem()

ae = IterativeAmplitudeEstimation(epsilon, alpha=alpha, quantum_instance=qi)

result = ae.estimate(problem)

conf_int = np.array(result.confidence_interval_processed)

print('Valor exacto:          \t%.4f' % exact_value)

print('Valor estimado:     \t%.4f' % (fixed_income.interpret(result)))

print('Intervalo de confianza:\t[%.4f, %.4f]' % tuple(conf_int))
```

Valor exacto:	2.1942
Valor estimado:	2.3567
Intervalo de confianza:	[2.3259, 2.3876]

3-. Pricing de las Option Call europeas:

Una opción de compra (option call) es un derivado financiero que otorga al comprador el derecho (pero no la obligación) de comprar en el futuro un activo al vendedor de la opción a un precio determinado previamente. Tenemos una Option Call europea K y un activo subyacente cuyo vencimiento ST sigue una distribución aleatoria dada La función de Payoff se define como:

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

A continuación, se utiliza un algoritmo cuántico basado en la estimación de amplitud para estimar la recompensa esperada, es decir, el precio justo antes de descontar, para la opción

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

lo que se traduce en: Δ, la derivada del precio de la opción con respecto al precio de contado, definida como:

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

La aproximación de la función objetivo y una introducción general a la fijación de precios de opciones y el análisis de riesgos en computadoras cuánticas se dan en los siguientes documentos.:

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

[1]: Como en los temas anteriores, comenzamos importando las librerías

```
import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np

from qiskit import Aer, QuantumCircuit

from qiskit.utils import QuantumInstance

from qiskit.algorithms import IterativeAmplitudeEstimation, EstimationProblem

from qiskit.circuit.library import LinearAmplitudeFunction
```

```
from qiskit_finance.circuit.library import LogNormalDistribution
```

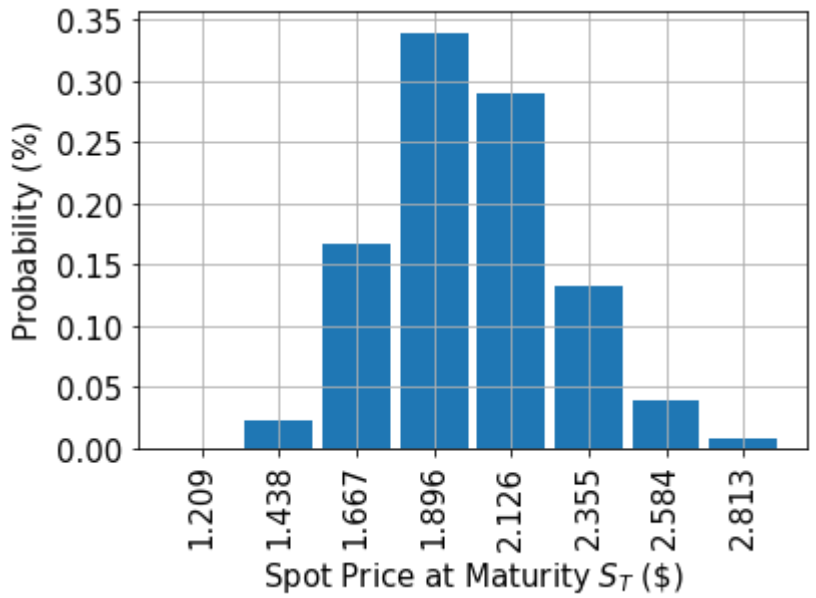
Modelo de incertidumbre

El modelo matemático de este apartado es igual que el del tema anterior, Renta fija. Aunque este enfoque es ligeramente distinto, primero, hay que tener en cuenta que poner precio a una Option Call es un poco más complejo que a un bono, ya que se trata de un producto complejo derivado, dicho esto, vamos a cambiar ciertos parámetros como, el número de qubits, ahora pasa a ser 3, y la distribución normal, ahora pasa a ser logarítmica, esto es debido a que el Option pricing se asemeja más en un entorno real, a una distribución logarítmica.

```
# número de qubits para representar la incertidumbre

num_uncertainty_qubits = 3

# parámetros para la distribución aleatoria considerada

S = 2.0        # precio de contado inicial

vol = 0.4      # volatilidad del 40%

r = 0.05       # tasa de interés anual de 4%

T = 40 / 365   # 40 días de madurez

# parámetros para la distribución logarítmica normal

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)

# el valor más bajo y más alto considerado para el precio al contado; en el
medio, se considera una discretización equidistante.

low  = np.maximum(0, mean - 3*stddev)

high = mean + 3*stddev

# componiendo el modelo de incertidumbre y el objetivo

uncertainty_model = LogNormalDistribution(num_uncertainty_qubits, mu=mu,
sigma=sigma**2, bounds=(low, high))
```

[1]: En este caso vamos plotear el modelo de probabilidad del precio al contado que tendrá en su vencimiento, como podemos observar, el valor que mayor probabilidad tiene está en torno a 1.896, veremos a continuación que no tiene porque ser el valor final calculado.

```
x = uncertainty_model.values

y = uncertainty_model.probabilities

plt.bar(x, y, width=0.2)

plt.xticks(x, size=15, rotation=90)

plt.yticks(size=15)

plt.grid()

plt.xlabel('Spot Price at Maturity $S_T$ (\$)', size=15)

plt.ylabel('Probability (\%)', size=15)

plt.show()
```



Función de Payoff

La función Payoff es igual a cero siempre que el precio al contado del vencimiento ST sea menos que el strike price K y sea creciente linealmente. La implementación usa un comparador, que cambia un qubit de |0⟩ to |1⟩ si $ST \geq K$, y este qubit se usa para controlar la parte lineal de la función de pago. La parte lineal en sí se aproxima de la siguiente manera. Explotamos el hecho de que $\sin 2(y+\pi/4) \approx y+½$ para un pequeño $|y|$. Por lo tanto, para un factor de reajuste de aproximación dado $c\text{approx} \in [0,1]$ y $x \in [0,1]$ consideramos

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

Podemos construir fácilmente un operador que actúe como usando rotaciones-Y controladas

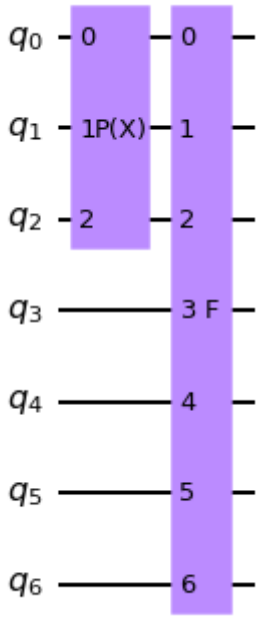
$$|x\rangle|0\rangle \mapsto |x\rangle \left(\cos(a * x + b)|0\rangle + \sin(a * x + b)|1\rangle\right),$$

Finalmente, nos interesa la probabilidad de medir |1⟩ en el último qubit, que corresponde a sin2(a∗x+b). Esto permite aproximar los valores de interés. Cuanto más pequeño elijamos $c$aprox, mejor será la aproximación. Sin embargo, dado que estamos estimando una propiedad escalada por $c$aprox, el número de qubits de evaluación debe ajustarse en consecuencia.

[2]: Vamos a realizar las siguientes acciones: establecemos el precio (debe estar dentro del valor low y high de la incertidumbre). establecemos la escala de aproximación para la función de pago. Configuramos la función objetivo lineal por partes en la variable breakpoints. Hacemos uso de la clase 'LinearAmplitudeFunction' que implementa una función lineal por partes en amplitudes de qubit. Es aquí donde volcamos nuestro modelo de incertidumbre calculado anteriormente.

```
strike_price = 1.896

c_approx = 0.25

breakpoints = [low, strike_price]

slopes = [0, 1]

offsets = [0, 0]

f_min = 0

f_max = high - strike_price

european_call_objective = LinearAmplitudeFunction(

   num_uncertainty_qubits,

   slopes,

   offsets,

   domain=(low, high),

   image=(f_min, f_max), breakpoints=breakpoints, rescaling_factor=c_approx)
```

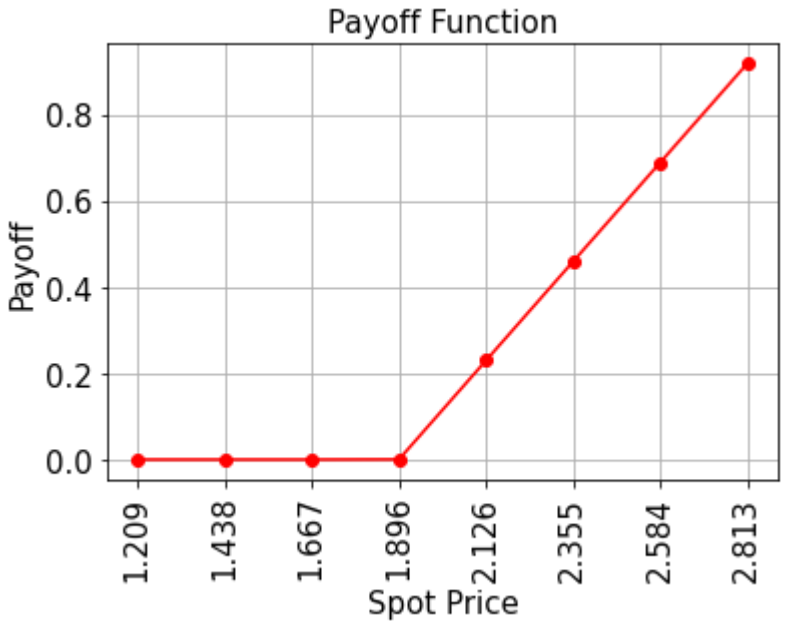
```
# construimos un operador para QAE (Quantum Amplitude Estimation) para la función
# de pago

# componemos el modelo de incertidumbre y el objetivo

num_qubits = european_call_objective.num_qubits

european_call = QuantumCircuit(num_qubits)

european_call.append(uncertainty_model, range(num_uncertainty_qubits))

european_call.append(european_call_objective, range(num_qubits))

european_call.draw()
```



[3]: ploteamos la función de pago exacta (evaluada en el modelo de incertidumbre)

```
x = uncertainty_model.values

y = np.maximum(0, x - strike_price)

plt.plot(x, y, 'ro-')

plt.grid()

plt.title('Payoff Function', size=15)

plt.xlabel('Spot Price', size=15)

plt.ylabel('Payoff', size=15)

plt.xticks(x, size=15, rotation=90)
```

```
plt.yticks(size=15)

plt.show()
```



[3]: evaluamos el valor esperado exacto (normalizado al intervalo [0, 1]). Delta es la relación que compara el cambio en el precio de un activo, generalmente valores negociables, con el cambio correspondiente en el precio de su derivado. Por ejemplo, si una opción sobre acciones tiene un valor delta de 0,8098, como en nuestro caso, esto significa que si el precio de la acción subyacente aumenta 1€ por acción, la opción aumentará 0,8098€ por acción, en igualdad de condiciones.

```
exact_value = np.dot(uncertainty_model.probabilities, y)

exact_delta = sum(uncertainty_model.probabilities[x >= strike_price])

print('valor esperado exacto:\t%.4f' % exact_value)

print('valor delta exacto:   \t%.4f' % exact_delta)
```

Payoff esperado exacto: 0.1623

Valor delta exacto: 0.8098

Evaluación del Payoff

En lugar de construir estos circuitos manualmente como hemos hecho, el módulo de finanzas de Qiskit ofrece la clase 'EuropeanCallPricingcircuito', que ya implementa esta función como bloque de construcción, además, implementa la clase 'Estimation Problem', encarga de hacer una estimación del payoff en base a nuestro modelo de incertidumbre.

```
from qiskit_finance.applications.estimation import EuropeanCallPricing

european_call_pricing                                                    =
EuropeanCallPricing(num_state_qubits=num_uncertainty_qubits,

                                          strike_price=strike_price,

                                          rescaling_factor=c_approx,

                                          bounds=(low, high),

                                          uncertainty_model=uncertainty_model)

# establecemos la precisión del objetivo y el nivel de confianza

epsilon = 0.01

alpha = 0.05

qi = QuantumInstance(Aer.get_backend('qasm_simulator'), shots=100)

problem = european_call_pricing.to_estimation_problem()

# construimos la estimación de la amplitud

ae = IterativeAmplitudeEstimation(epsilon, alpha=alpha, quantum_instance=qi)

result = ae.estimate(problem)

conf_int = np.array(result.confidence_interval_processed)

print('Valor exacto:        \t%.4f' % exact_value)

print('Valor estimado:    \t%.4f' % (european_call_pricing.interpret(result)))

print('Intervalo de confianza:\t[%.4f, %.4f]' % tuple(conf_int))
```

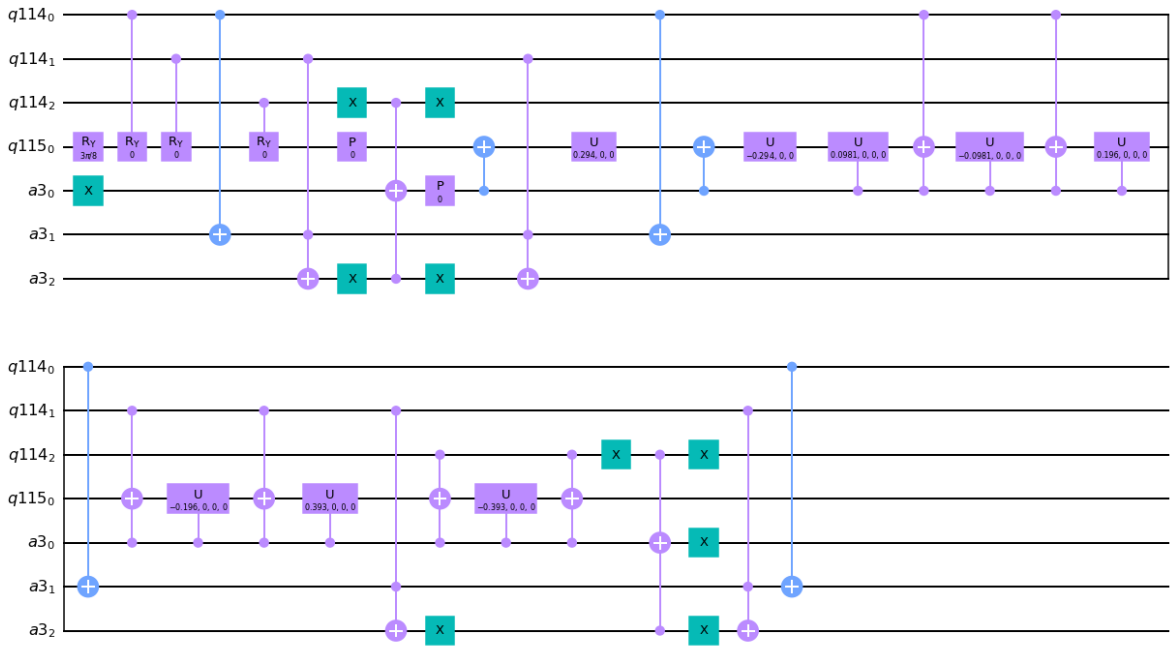
Valor exacto: 0.1623

Valor estimado: 0.1673

Intervalo de confianza: [0.1623, 0.1722]

[4]: Mostramos como sería el circuito que implementa dicha clase, se observan rotaciones sobre el eje Y varios qubits de control

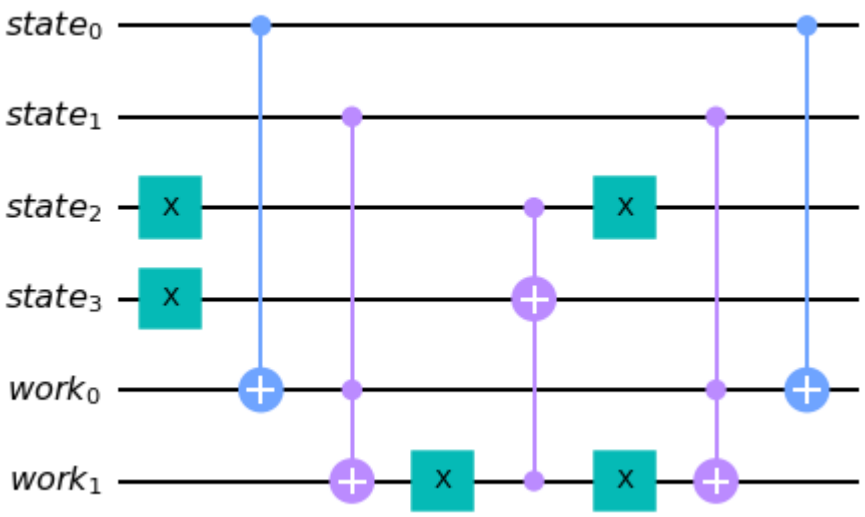
```
european_call_pricing._objective.decompose().draw()
```



Evaluación del Delta

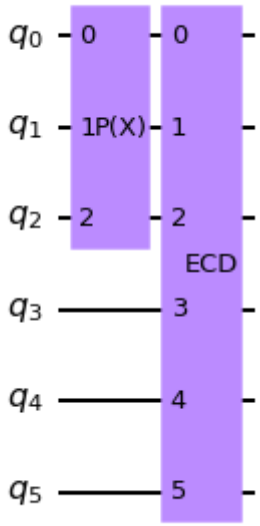
Dado que solo nos interesa la probabilidad de que esta condición sea True, podemos usar directamente un qubit de control como el qubit objetivo en la estimación de amplitud sin ninguna aproximación adicional. El Delta es un poco más simple de evaluar que el Payoff. Usamos un circuito comparador y un qubit de control para identificar los casos en los que ST>K

```
from qiskit_finance.applications.estimation import EuropeanCallDelta

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

european_call_delta._objective.decompose().draw()
```




```
european_call_delta_circ                                                    =
QuantumCircuit(european_call_delta._objective.num_qubits)

european_call_delta_circ.append(uncertainty_model, range(num_uncertainty_qubits))

european_call_delta_circ.append(european_call_delta._objective,
range(european_call_delta._objective.num_qubits))



european_call_delta_circ.draw()
```

Out[9]:




```
# establecemos la precisión del objetivo y el nivel de confianza

epsilon = 0.01

alpha = 0.05
```

```
qi = QuantumInstance(Aer.get_backend('qasm_simulator'), shots=100)

problem = european_call_delta.to_estimation_problem()

# construir estimación de amplitud

ae_delta = IterativeAmplitudeEstimation(epsilon, alpha=alpha,
quantum_instance=qi)

result_delta = ae_delta.estimate(problem)

conf_int = np.array(result_delta.confidence_interval_processed)

print('Delta exacto:    \t%.4f' % exact_delta)

print('valor estimado: \t%.4f' % european_call_delta.interpret(result_delta))

print('Intervalo de confianza: \t[%.4f, %.4f]' % tuple(conf_int))
```

Delta exacto: 0.8098

Valor estimado: 0.8092

Intervalo de confianza: [0.8052, 0.8133]

Código C++ clásico:

```
#define _USE_MATH_DEFINES

#include <iostream>

#include <cmath>

double norm_pdf (const double & x) {

    return (1.0 / (pow (2 * M_PI, 0.5))) * exp (-0.5 * x * x);

}

double norm_cdf (const double & x) {

    double k = 1.0 / (1.0 + 0.2316419 * x);

    double k_sum = k * (0.319381530 + k * (- 0.356563782 + k * (1.781477937 + k *
(- 1.821255978 + 1.330274429 * k))));

    if (x> = 0.0) {
```

```
            return (1.0 - (1.0 / (pow (2 * M_PI, 0.5))) * exp (-0.5 * x * x) *
k_sum);

    } else {

        return 1.0 - norm_cdf (-x);

    }

}

double d_j (const int & j, const double & S, const double & K, const double & r,
const double & v, const double & T) {

     return (log (S / K) + (r + (pow (-1, j-1)) * 0.5 * v * v) * T) / (v * (pow
(T, 0.5)));

}

double call_price (const double & S, const double & K, const double & r, const
double & v, const double & T) {

     return S * norm_cdf (d_j (1, S, K, r, v, T)) - K * exp (-r * T) * norm_cdf
(d_j (2, S, K, r, v, T));

}

int main (int argc, char ** argv) {

    double S = 100.0; // Option price

    double K = 100.0; // Precio de ejercicio

    double r = 0.05; // Tasa libre de riesgo(5%)

    double v = 0.2; // Volatilidad del subyacente (20%)

    double T = 1.0; // Un año hasta el vencimiento



    double call = call_price (S, K, r, v, T);

     std :: cout << "Call Price:" << call << std :: endl;

    return 0;

}
```

[5]: Vamos a exponer las diferencias a tener en cuenta entre este tema en Qiskit y C++. El código en C++ hace solo el cálculo de la distribución normal, la aproximación para la desviación estandar, y el Payoff, dado que el Delta se calcula de una forma muy similar, además, como ocurre en el tema anterior de Renta Fija, la función cuántica de incertidumbre es más completa y realista que la aplicada de forma clásica, con una distribución normal y aproximación con parámetros manualmente declarados.

Bibliografía:

-https://qiskit.org/documentation/finance/tutorials/11_time_series.html
-https://qiskit.org/documentation/stubs/qiskit.finance.data_providers.RandomDataProvider.html
-https://qiskit.org/documentation/_modules/qiskit/finance/data_providers/wikipedia_data_provider.html
-https://es.finance.yahoo.com/
-https://www.quandl.com/
-https://qiskit.org/documentation/stubs/qiskit.finance.data_providers.DataOnDemandProvider.html
-https://qiskit.org/documentation/stubs/qiskit.circuit.library.NormalDistribution.html
-https://qiskit.org/documentation/finance/stubs/qiskit_finance.applications.FixedIncomePricing.html
-https://qiskit.org/documentation/stubs/qiskit.aqua.algorithms.IterativeAmplitudeEstimation.html
-https://qiskit.org/documentation/stubs/qiskit.circuit.library.RYGate.html
-https://qiskit.org/documentation/stubs/qiskit.circuit.library.LinearAmplitudeFunction.html
-https://blog.ibroker.es/opciones-financieras-el-desafio-de-la-delta-la-batalla-de-la-vega-parte-xvii-lunes-22-de-junio-2020/
-https://optionstradingiq.com/call-option-payoff-graph/
-https://qiskit.org/documentation/finance/stubs/qiskit_finance.applications.EuropeanCallPricing.html
-https://qiskit.org/documentation/stubs/qiskit.algorithms.EstimationProblem.html
-https://www.quantstart.com/articles/European-vanilla-option-pricing-with-C-and-analytic-formulae/