# Quantum Programming for Market Risks

V. Bouza, I. Marteens, J. Nebrera

*Comunytek Consultores*

**ABSTRACT**

As Quantum Computing grows in popularity, more research is conducted to find areas which could benefit from this new technology. This paper presents the experiences gained from a proof-of-concept project intended to implement a Value at Risk algorithm in a quantum computer.

*Keywords*: quantum programming, market risk, composability

## Contents

1 Introduction ....... 3
2 The Quantum Computing Paradigm ....... 3
2.1 Tension with Classical Computing ....... 4
3 Quantum Composability ....... 4
3.1 Randomness ....... 5
4 Quantum Computing versus the Von Neumann Architecture ....... 5
5 The Probabilistic Model ....... 5
5.1 Normally distributed prices ....... 6
5.2 Logarithmic returns ....... 6
5.3 Diagonalization ....... 7
5.4 Approximation using the Taylor's expansion ....... 8
6 Implementation ....... 8
6.1 Grover-Rudolph's state initialization ....... 8
6.2 Amplitude Estimation ....... 8
7 Conclusions ....... 9
8 References ....... 10

# 1 Introduction

As Quantum Computing grows in popularity, more research is conducted to find areas which could benefit from this new technology. The required skills for programming a quantum computer, nevertheless, can be daunting.

This paper presents the experiences gained from a proof-of-concept project intended to implement a Value at Risk algorithm in a quantum computer.

# 2 The Quantum Computing Paradigm

Most courses and textbooks explain quantum computing in a bottom-up fashion, starting with circuits and combining them. We will start, on the contrary, by presenting an alternative top-down approach, in the hope that this view could offer insights on how much can achieve with our current hardware.

A quantum computer can be abstracted as a system with a state represented by a vector $\psi$ in $\mathbb{C}^{2^N}$, satisfying the condition $\|\psi\| = 1$, where the double bars represents the length of the vector. The $N$ in the vector space dimension corresponds to the number of qubits the computer is made of. Several alternative representations can be used, such as the density matrix or a projective space, but we will stick to the first definition for the sake of simplicity.

Now that we have defined the scenario, let's explain the dynamics of the system:

1. The quantum computer always starts from a priviledged vector in the underlying vector space.

   Actually, a quantum computer always have a distinguished basis, which is called the *computational basis*, and this initial vector is just one of the vectors from that basis. The distinguished basis is determined by the measurement process that will take place in the third step of this list of actions.

2. The state of the system is somehow moved to a given point in the hypersphere.

   Since we are dealing with points in a complex hypersphere, this transformation amounts to a rotation of the state vector. The important thing here is the final state: it doesn't matters how we reach that state. As a matter of fact, the rotation is achieved incrementally, as we will see soon.

3. A *measurement* takes place.

   The measurement is an irreversible process that moves the state into one of the vectors of the computational basis. We cannot predict which vector will be choosen, but only the probability of each possible result.

We can graphically summarize this process as dragging a point on the surface of a complex hypersphere, dropping it and watching how it moves to one of a series of predefined marks in that hypersurface.

The first important point of all this imagery is that the path used when dragging the state is of no consequence for the outcome of this procedure. A quantum algorithm is completely defined by a point in the $\mathbb{C}^{2^N}$ projective space or, alternatively, by the unitary operator $U$ that rotates the initial state into the state before measurement.

A second detail to notice is that a quantum algorithm never hangs, at least in its measurement phase. The *physical* process of measurement is, up to our best knowledge, instantaneous. So, once you have told which is your algorithm, you can expect an answer in the time required by the control circuit to display the results.

The third point to remark is how laconic are answers from a quantum oracle, compared to the plethora of possible questions. A computer with $N$ qubits can only give one from $2^N$ answers, while the set of possible algorithms is an uncountable one. Those answers, on the other hand, are not deterministic. So, some problems require probably will more than a run of the algorithm.

### 2.1 Tension with Classical Computing

This paradigm is in stark contrast with the classical one. We have seen that the measurement phase always suceeds in a negligible amount of time. How about the implementation of the unitary transform? One of the consequences to the Solovay-Kitaev algorithm [3] is that any unitary matrix can be implemented efficiently with a given finite set of quantum gates. So, all single-measurement algorithm in a quantum computer are known to halt. It is hard to reconcile the existence of semialgorithms in the classical realm with the lack of them in this model of quantum computing.

Actually, what our current model of quantum hardware implements is a finite implementation of an abstraction known as Quantum Finite Automata. The capacities of this abstraction is still an open problem in Computer Science.

## 3 Quantum Composability

Fortunately, calculating market risks using the quantum paradigm is not a problematic task. So, we will focus, from now on, in how quantum algorithms can be designed.

Composability, the ability to combine two or more existing algorithms into a bigger one, was hard to achieve in the Von Neumann architecture.This is a paradox. We had to add several layers of software abstractions to the bare hardware to achieve this goal.

On the other hand, quantum computers allows a more direct approach to composability. And, at the same time, this approach is more limited.

Recapitulating, most practical quantum algorithms can be divided in these stages:

1. Initialisation of the quantum state.
2. Unitary evolution of the state (our previously explained *drag* phase).
3. Measure (the instantaneous *drop* phase).

Composability plays a role in quantum programming in two forms. The most obvious is the high-level one: composition as matrix multiplication. We can stack transformations on a common state or sub-state.

There's a second role for composability and it is not so obvious. It has to do with how efficiently a unitary transformation can be implemented using quantum circuits. So, if we need to perform a transformation, it is a plus if that transformation can be decomposed in a sequence of easily implementable circuits.

### 3.1 Randomness

The most counterintuitive part about quantum programming is randomness. God plays dice, but only when the party is over. The unitary evolution of quantum state is fully deterministic. Randomness appears only at the end, on measure, when information is destroyed.

This unavoidable fact has consequences for all classical algorithms that start by drawing a sample from a random number generator. There is no direct translation for them. However, we still have two or three tricks under the sleeve.

The first trick could be the easiest one: allow *weak measures* on the quantum system. We could measure the state of part of the bits and then inject this information as part of the initialization of another subsystem. Regretfuly, this is not allowed by state-of-the-art quantum computers.

The second trick starts by accepting that randomness can only be introduced in the last step of the algorithm. We can then prepare the state in reverse, so that the final throwing of the dice completes our quest. There is little to none literature on this topic.

The third trick is the one we followed in this proof of concept. We start by initializing a quantum state with a superposition of all vectors from the computational base like in the Grover algorithm initialization. However, instead of asigning a uniform probability for all the vectors, a normal distribution is applied by initializing the amplitude of each superposed state accordingly to the normal distribution histogram.

## 4 Quantum Computing versus the Von Neumann Architecture

It is useful to compare the architecture of a quantum computer with the classical Von Neumann architecture.

1. Data and code are no longer stored both in memory.
2. There is a clear separation between data, as stored in qubit states, and data stored as parameters of some circuits. For instance, you can rotate the state using a simple rotation transform, but the rotation angle must be hardcoded into the rotation gate.
3. Data is not easily addressable from “code”, i.e., circuits.
4. There are no loops. Do you want to apply a transformation twice? Concatenate two instances of the circuit corresponding to the transformation.
5. All calculations are reversible.
6. Quantum information cannot be cloned.

Even state initialization must be performed by circuits since all qubits starts in one of the vectors of the base.

## 5 The Probabilistic Model

We start with a portfolio with $N$ assets, and historical prices for these assets for a homogeneus set of dates. Each asset is weighted inside the portfolio using a vector $|w\rangle$. If the vectors of prices is represented as $|v\rangle$, the net value of the portfolio is the dot product of these two vectors:

$$X = \langle w|v\rangle$$

Now we want to calculate the Value at Risk for a given confidence level $\alpha$, which is defined like this:

$$VaR_\alpha = \inf\{x : P(X \geq x) = \alpha\}$$

Now we can take one of two approaches for calculating the Value at Risk:

1. Assume that each asset price follows a normal distribution rule.
2. Find daily returns for each price series, take the logarithm, and assume that those transformed series are the ones that follow a normal distribution rule.

Please note that the normality hypothesis is only required if we want to generate samples drawn from a normal distribution generator. As an alternative, for instance, we could have used a Johnson's $S_U$-distribution, which can be characterized by four parameters, to take skewness and kurtosis into account.

## 5.1 Normally distributed prices

Let $|\mu\rangle$ contain the mean of each asset in the portfolio and let $\Sigma$ be the covariance matrix. The mean and the variance of the linear combination of variables used for calculating the price can be calculated using these formulas:

$$\mathbf{E}(X) = \langle w|\mu\rangle$$
$$\sigma_X^2 = \langle w|\Sigma|w\rangle$$

It is very important to note that the theorem linking the mean and variance of a linear combination of random variables does not assumes any kind of distribution for the individual variables. However, if we add the normality hypothesis, then the distribution of $V$ is also normal.

Having gone this far, it makes no sense to sample from the distribution of $V$, since we can immediately calculate the VaR using the data at hand:

$$\mathbf{E}(X) - probit(\alpha)\sigma_V X_0$$

where $X_0$ is the current value of the portfolio and $probit(\alpha)$ is the quantile function associated with the normal standard distribution.

## 5.2 Logarithmic returns

The previously sketched method makes a bold prediction. Given the past performance of the portfolio, we dare to predict how bad could be our losses. The *boldness* of the statement has mainly to do with the time ahead for the forecast to come true.

We could play safer by guessing how much money we can lose in a fixed number of days. Central counterparties, for instance, charge their members with the estimated amount that could be lost on a close-out period of five days. For the sake of simplicity, we will assume that we want to estimate our loss from one day to the next.

Let $V_i = \{\, v_{ij} : i \in 1 \dots N,\ j \in 1 \dots m\}$ be the price series for each asset. We have $N$ assets and $m$ prices for each asset, all of them drawn from a homogeneous set of dates. What we need are these derived series:

$$R_i = \{\, r_{ij} : i \in 1 \dots N, j \in 1 \dots m-1, r_{ij} = \ln(v_{i,j+1}/v_{ij})\}$$

Then, we can suppose these logarithmic returns are normally distributed. As before, we will use $|\mu\rangle$ for the means and $\Sigma$ for the covariance matrix.

This time, however, there is no easy method to convert the multivariate distribution into a univariate. Every time we draw a vector $|r\rangle$ from the multivariate distribution, we are predicting this price for the whole portfolio:

$$X = \sum w_i v_{i,0}\, e^{r_i}$$

There is no simple formula for the joint distribution, so we are forced to deal with a multivariate normal distribution.

The standard procedure for generating a multivariate normal distribution requires a generator for a standard normal distribution. Let us forget, for now, how we might generate a standard normal vector $S$. For a general multivariate normal, we need to stretch and move the standard normal vector. Moving the vector is achieved by adding the mean of the desired distribution. We will call $M$ that mean value. If we were to stretch a standard univariate normal into a general one, we would multiply the original variable by the standard deviation, $\sigma$. Since we have a multivariate distribution, we need the square root of the covariance matrix, which must be calculated by applying the Cholesky decomposition to the original covariance matrix:

$$\Sigma = L^T L$$
$$R = M + L \times S$$

### 5.3 Diagonalization

Regardless the scheme we choose for sampling, multiplying by a Cholesky matrix is plain bad news. If we encode values as base vectors, we need to perform $n(n+1)/2$ multiplications. And if we encode values as amplitudes to be transformed, we just step into the fact that the Cholesky matrix is a triangular matrix, not a unitary operator.

Luckily for us, there is a simple trick to remove Cholesky from the picture. We can always decompose a properly designed covariance matrix using and orthogonal matrix created by its eigenvector and a diagonal matrix containing the corresponding eigenvalues:

$$\Sigma = QDQ^{-1}$$

$Q$ contains all the eigenvectors from $\Sigma$ as columns, and $D$ is a diagonal matrix formed by the eigenvalues. So, we can rotate our coordinates system using the orthogonal operator and draw random samples using a simpler formula:

$$R' = M' + \sqrt{D} \times S$$

Please, note that:

1. The square root of a diagonal matrix is another diagonal matrix.
2. The matrix product has been transformed in a much more efficient dot product.
3. The vector with mean values has been also rotated.

We still need to convert prices back to the original set of coordinates, using the inverse of the rotation. But this conversion is only required at the end of the algorithm and, in any case, an orthogonal matrix is also a unitary operator, easier to implement by a quantum circuit.

### 5.4 Approximation using the Taylor's expansion

When returns are small enough, we can use the first two terms of the Taylor series to get an approximate value of the random variable $X$:

$$X = \sum w_i v_{i,0}\,(1 + r_i) = X_0 + \sum w_i v_{i,0}\, r_i$$

Since we have assumed that the $r_i$ variables are drawn from a normal distribution, we have here again a linear combination of normal variables, and $X$ can be reduced, as before, to a univariate standard random variable.

## 6 Implementation

The full algorithm implemented as part of our project is presented and explained elsewhere. We will only include here an outline of it, for completeness.

### 6.1 Grover-Rudolph's state initialization

Our algorithm starts by creating a weighted superposed state with all vectors from the computational base, using a scaled normal distribution for calculating the weights:

$$|\psi\rangle = \sum \sqrt{p_i}|i\rangle$$

Here, $|i\rangle$ represents each of the $2^N$ states from the computational base vectors. The problem we face is that, even such a simple abstraction as initializing qubits in a known state, it is not supported by current quantum hardware. Qubits are always initialized as in the base state $|0\rangle$. Thus, we will first apply a Walsh-Hadamard transform to the initial state, to create a uniformly superposed state:

$$H^{\otimes N}|00\ldots0\rangle = \sum |i\rangle$$

Then, we apply this operator to convert this state into the weighted state we need:

$$W = \sum \sqrt{p_i}|i\rangle\langle i|$$

The corresponding circuit is implemented by a chain of controlled rotations, one qubit at a time [4].

### 6.2 Amplitude Estimation

Once we have modeled the distribution of a random variable, we can use Amplitude Estimation [2] to get estimators based on the distribution. We need an additional function $f\colon 0..N-1 \rightarrow [0,1]$ and an operator performing this transformation:

$$F : |i\rangle_n|0\rangle \;\rightarrow\; |i\rangle_n(\sqrt{(1-f(i)}|0\rangle + \sqrt{f(i)}|1\rangle)$$

When we apply $F$ to the state initialized with our random distribution, we obtain this state:

$$\sum \sqrt{(1-f(i)}\sqrt{p_i}|i\rangle_n|0\rangle + \sum \sqrt{f(i)}\sqrt{p_i}|i\rangle_n|1\rangle$$

Amplitude Estimation allows us to estimate the amplitude of the second term in the above sum using additional qubits and phase estimation. After squaring amplitudes, this is what we get:

$$\sum f(i)p_i = \mathbb{E}[f(X)]$$

Thus, we can decide what we estimate by changing the definition of $f$.

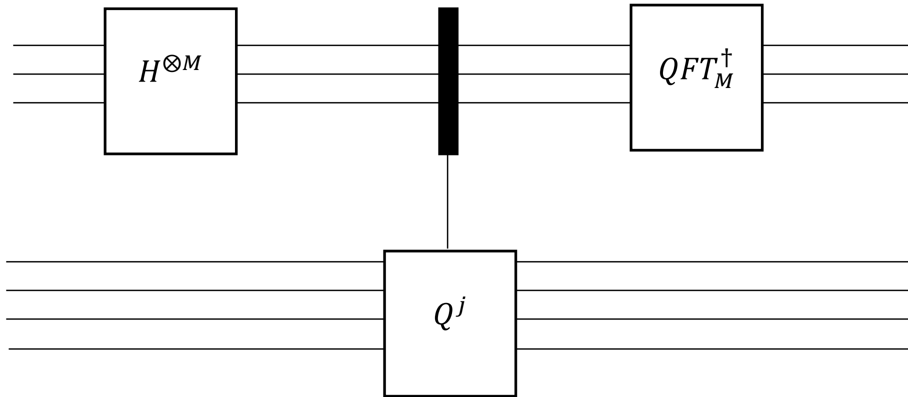
Let's call $A$ this composite operator:

$$A: |i\rangle_n|0\rangle \rightarrow \sum \sqrt{(1-f(i)}\sqrt{p_i}|i\rangle_n|0\rangle + \sum \sqrt{f(i)}\sqrt{p_i}|i\rangle_n|1\rangle$$

We then define the following operator to be used by Amplitude Estimation in the usual manner:

$$Q = -AS_0AS_{\psi_0}$$

AE itself is implemented by the following circuit:




The $QFT_M^\dagger$ is the inverse Quantum Fourier Transform acting on the $M$ qubits allocated for the estimated amplitude.

## 7 Conclusions

Quantum programming faces two obstacles for a wider adoption, besides the current lack of reliable hardware:

1. Composability is poor, mainly for the lack of proper abstractions. Quantum algorithms are still too close to the metal.
2. Since true randomness can only be introduced at the final of the computations, at least with the available computers, it is complex to design Monte Carlo simulations that operates with non-uniform distributions.

The trick performed by the Grover-Rudolph's state initialization is a theoretical step forward, but is not enough to overcome the second hurdle for practical purposes. Though this initialization allows a non-uniform superposition of states, as in the original Grover algorithm, the states are encoded as binary numbers by the vector base, and further processing of this encoding is out of reach for our current hardware.

Even in our toy project, we had to perform two related simplifications: downgrading from a multivariate distribution to a univariate one, and use the linear part of the Taylor expansion to avoid scaling and exponentiation of the state encoding.

We think that the most promising way to move forward is to accept the fact that true randomness only appears in the last step of any quantum computation and perform more research on how design algorithms in a "reverse" order, starting from what we want to obtain as answer, and the finding the required transformations from last to beginning.

## 8 References

[1] Bennet, C. H., Bernstein, E., Brassard, G., and Vazirani, U. V. 1997. Strengths and weaknesses of quantum computing. SIAM J. Comput. 26, 1510–1523.

[2] Brassard, G., Hoyer, P., Mosca, M. & Tapp, A. Quantum amplitude amplification and estimation. Contemporary Mathematics 305, https://doi.org/10.1090/conm/305 (2002).

[3] Dawson, Christopher M.; Nielsen, Michael 2006. The Solovay-Kitaev algorithm. Quantum Information & Computation. arXiv:quant-ph/0505030

[4] Lov Grover and Terry Rudolph (2008), Creating superpositions that correspond to efficiently integrable probability distributions, arXiv:quantph/0208112v1 15 Aug 2002

[5] Shor, P. W. 2003. Why haven't more quantum algorithms been found? Journal of the ACM, Vol. 50, No. 1, pp 87-90.