Quantum algorithmic differentiation

Giuseppe Colucci[a,b] and Francesco Giacosa[a,c]

[a]*Institute of Physics, Jan Kochanowski University, ul. Uniwersytecka 7, 25-406 Kielce, Poland*
[b]*De Volksbank N.V., Croeselaan 1, 3521 BJ Utrecht, The Netherlands* [1]
[c] *Institute for Theoretical Physics, Goethe University, Max-von-Laue-Str. 1, 60438 Frankfurt am Main, Germany*

Abstract

In this work we present an algorithm to perform algorithmic differentiation in the context of quantum computing. For the derivative of the arbitrary composite function $f \circ g \circ ... \circ h$, three n-qubit states are sufficient, where n refers to the quantum computer precision. Namely, with these 3 n-qubit states we construct a set of operation which can be iteratively applied in such a way to calculate the derivative of any composite function. For the sum, $f(x) + g(x) + ... + h(x)$ or the product $f(x) \cdot g(x) \cdot ... \cdot h(x)$ one needs at least 6 n-qubit states, if a re-initialization is implemented. Since the implementation of elementary functions is already possible on quantum computers, the scheme that we propose can be easily applied. Moreover, since some steps (such as the CNOT operator) can (or will be) faster on a quantum computer than on a classical one, the applied procedure may lead in the (near or far) future to quantum algorithmic differentiation being more advantageous than its classical counterpart.

---

[1]The analysis, views and opinions presented in this paper are our own and do not represent the opinions of any firm.

1 Introduction

Algorithmic differentiation (also known as *automatic* or *computational* differentiation, hereafter AD) [1, 2, 3] has gained particular attention in the last years due to its practical use in finance, in particular in the context of pricing financial derivatives [4, 5], as well as in data science and machine learning [6].

In finance, AD is used to compute sensitivities (or Greeks) of the price of financial instruments with respect to the underlying drivers both accurately (to machine precision) and efficiently [7, 8, 4]. The advantage of AD w.r.t. the standard numerical differentiation is particularly evident when dealing with complex financial instrument which require numerical pricing methods, e.g. Monte Carlo. In fact, the calculation of price sensitivities with numerical differentiation involves perturbing the underlying price drivers in turn, repeating simulations and getting finite-difference approximations. The computational effort thus becomes very significant when the financial instrument has a large number of drivers, as is typically the case for the sophisticated models for which Monte Carlo simulations are used in practice.

In data science, a particular case of algorithmic differentiation, known as *backpropagation*, is used in Artificial Neural Network training to calculate gradients of the loss function with respect to the weights of the network ([6] and refs. therein). With the advance of deep learning, the need of increased efficiency and precision in the calculation of the gradients arose due to the large number of layers used to build up the networks.

Currently, several attempts have been made to introduce the concept and framework of differentiation to the quantum computing world [9, 10, 11, 12]. In fact, the progresses made in creating optimization, simulations [13, 14], financial applications [15, 16], quantum chemistry [11], and machine learning algorithms [17, 18, 19, 20] on a quantum computer require the calculation of derivatives.

Until now, however, to the best of our knowledge no algorithm for AD has been proposed in the context of quantum computation. However, the advances in scientific computing algorithms for quantum computers [21, 22, 23, 24, 25, 26, 27] allow for the definition of primitive functions which can be used for algorithmic differentiation. Inspired by this approach, we propose in this paper a framework to implement algorithmic differentiation on a quantum computer, the *quantum algorithmic differentiation framework*. Two versions are proposed: a fully quantum version, with no need of a hy-

brid quantum-classical system, and one in which classical ancilla registers are used for a more practical implementation. The proposed framework allows for the calculation of a composite function and its derivative in a given point at *quantum* machine precision. The precision of the quantum algorithmic differentiation depends on the number of available qubits to represent a floating number on the quantum computer. This currently poses difficulties in implementing this algorithm (as well as many other currently proposed algorithms) since current quantum hardware development does not allow for large qubits. Nevertheless, this paper is intended to extend the available algorithms of scientific calculation on quantum computer and prompt possible extension of optimization and machine learning quantum algorithms with quantum algorithmic differentiation.

As we shall explain in the text, the minimal number of qubits needed to evaluate the derivative of an arbitrary composed function of elementary functions amounts $3n$, where n is the quantum computer precision. When including the sum and/or the products of elementary functions, the minimal number of needed qubits is $6n$, if a re initialization procedure is employed.

The paper is structured as follows: in sec. 2 the classical algorithmic differentiation is introduced. Sec. 3 presents a translation of the classical framework for algorithmic differentiation to the quantum case. Here the operators and procedures to calculate a function and its derivative are introduced. Sec. 4 formalizes the results from the previous section and outlines the algorithm to calculate the derivative of a composite function at a given point (represented by a quantum state of n-qubits). Finally in Sec. 5 we summarize our findings and outline possible future developments.

2 Algorithmic differentiation

Algorithmic differentiation (AD) [2, 8] uses exact formulas along with floating-point values to calculate the derivative of a composite function starting from known elementary (or primitive) functions. It involves no approximation error as in numerical differentiation. AD is a third alternative to symbolic and numerical differentiation, and is also called computational differentiation or automatic differentiation.

The basic idea behind AD is to calculate the derivative of each a primitive function in an iterative way, and store at each step a tuple containing the value of the function and of the derivative, (val, der). For sake of simplicity

3

we shall call this tuple the *valder* object.

More precisely, AD corresponds to the computational implementation of the chain rule. In fact, by applying the chain rule repeatedly to primitive functions (and arithmetic operations), derivatives of arbitrary order of composite functions can be computed automatically.

An example can be used to clarify it. Consider the function $f(x) = x^2 \cdot \sin(\log(x))$. Assume we want to calculate the derivative of this function at x_0. Let us consider the *valder* object $(x_0, 1)$ to input our algorithm.

The function $f(x)$ can be rewritten as:

$$
\begin{aligned}
f(x) &= u(x) \cdot h(\log(x)) & (1) \\
&= u(x) \cdot h(w(x)). & (2)
\end{aligned}
$$

We can now split and execute the calculation of the derivative of f by calculating the derivatives of each component, $c(x)$. The algorithm is based on a series of iterations that take as input a *valder* object (v, d) and define a new *valder* object as $(c(v), c'(v) \cdot d)$, where $c(x)$ is the component we are looking at.

For the function $f(x)$ we have:

- define the input *valder* tuple $(x_0, 1)$.

- for $u(x)$:

  - calculate the derivative of u at x_0 and define a new *valder* object containing the value of the function $u(x_0)$ and derivative $u'(x_0) \cdot 1$ (where 1 is the input value from the input tuple).
  - store the *valder* tuple : $(u(x_0), u'(x_0)) = (val_1, der_1)$

- for $h(w(x))$:

  - calculate the values of $w(x_0)$ and $w'(x_0)$.
  - define the *valder* tuple $(w(x_0), w'(x_0) \cdot 1) = (y, d)$ to be the input for the next step.
  - calculate the values of the function $h(y)$ and derivative $h'(y)$.
  - define the *valder* tuple $(h(y), h'(y) \cdot d) = (val_2, der_2)$

- Finally calculate the derivative of the product from the two obtained tuples as:
$$(val_1 * val_2, val_1 * der_2 + val_2 * der_1).$$

This example shows that from the knowledge of primitive functions and their derivatives we can calculate the derivative of a complex function without the error associated to numerical differentiation.

It can be shown that for practical applications the following primitive functions are enough to handle most typical situations: $exp(u)$, $log(u)$, $sqrt(u)$, $sin(u)$, $cos(u)$, $tan(u)$, $asin(u)$, $atan(u)$. Further the following operations are needed: $plus(u,v)$, $minus(u)$, $times(u,v)$, $reciprocal(u)$.

For all of the above mentioned primitive functions an implementation in terms of quantum circuits already exists [24, 26, 27]. Therefore in the following we develop a method to perform AD on a quantum computer.

For sake of simplicity we shall focus on the algorithmic differentiation in the *forward mode*. See [6] for a review of the forward and reverse mode implementations of algorithmic differentiation.

3 Quantum algorithmic differentiation of a composite function

Consider three n-qubits states, $|s\rangle$:

$$|s\rangle = |s_m\rangle \otimes |s_{m-1}\rangle \otimes ...|s_0\rangle \otimes |s_{-1}\rangle \otimes ... \otimes |s_{m-n}\rangle \tag{3}$$

The first m (where $m < n$) terms are the integer part and the last $n - m$ terms are the fractional part of a number.

Let v be a number in $\mathbf{R}$, and $|v\rangle$ the n-qubit representation of the number with precision $\frac{1}{2^{n-m}}$.

Assume we want to calculate the derivative of a function $f(x)$ at x_0.

The building blocks of the quantum algorithmic differentiation are the following:

- *Transfer* or *Copy* operator, $\mathcal{C}$: a series of CNOT between the first and second n-qubit states.

- *Reset* procedure, $\mathcal{R}$: a procedure to reset the second n-qubit to 0 (as a value, therefore $|s\rangle = |00...0\rangle$).

- $\mathcal{AD}(f)$: this operator calculates the value of a function and its derivative at a point v, respectively $f(v)$ and $f'(v)$, for a set of known functions (e.g. $\sin, \log, \exp$, etc.).

In the following we describe the outlined building blocks.

5

Transfer or *Copy* operator

The transfer (or copy) operator is an extension of the controlled-Not (CNOT) operator to the tensor product of two qubits. Such operator acts as a copy, or transfer, of the controlled qubit onto the target qubit, with the condition that the latter state is set to $|0\rangle$ (or $|00...0\rangle$). In case the state $|s\rangle$ is built up of n-qubits:
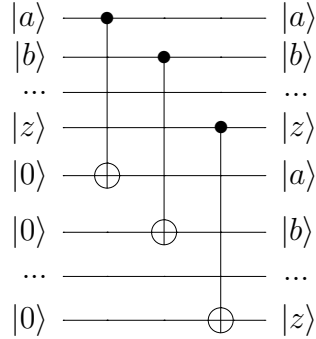
for any a, b and $z = 0, 1$.

In fact, in the proposed algorithm the control qubit will correspond to the value of a function at a point, while the target will be set to 0. Therefore the goal of the transfer operator will be to copy the value of the function at the point onto the second qubit.

Quite interestingly, there are ways to realize a fast CNOT obtained by employing Quantum Zeno dynamics [28], thus possibly rendering the whole procedure that we are about to describe more efficient and possibly even faster than a standard classical computation.

Reset procedure

In order to use the described transfer operator, the second n-qubit state needs to be set to zero ($|0\rangle$ or $|00...0\rangle$). After the application of the transfer operator, such state will be in general in a non-zero state, therefore a procedure to reset such state to zero is needed.

In general, one possibility would be to revert the unitary operations applied on the state. However, this could affect also states that we do not necessarily want to reset if operators across tensor space of several qubits are used. The following two possibilities are therefore considered:

- Fully quantum implementation: prepare as many ancilla zero states ($|0\rangle$ or $|00...0\rangle$) as needed for the calculation. Whenever needed to reset the

state, use a *Swap gate* to swap the non-zero qubit with a state set to zero.

- Hybrid implementation: use a single classical bit c_0. Perform a measure on the standard Z-basis and measure it on c_0. Apply a bit-flip operator (multidimensional extension of σ_X). If the classical bit is zero apply again a bit-flip operator. The qubit is then back in the zero state.

For sake of readability, in the following we will indicate the procedure with $\mathcal{R}$, without specifying which of the the two procedures is applied.

$\mathcal{AD}(\cdot)$ operators

The operator $\mathcal{AD}(f)$ is associated to a particular primitive function (e.g. sin or log). Implementations for such functions are available in [27] and ref. therein. For each function $f(x)$, the operator takes as input three n-qubit states, $|a\rangle, |b\rangle, |c\rangle$ and outputs a state with the following 3 n-qubit states: $|f(a)\rangle, |f'(b)\rangle, |f'(b) * c\rangle$. The operator $\mathcal{AD}(f)$ is built upon two operators: the first, $f \otimes f'$, is represented by a multigate, in which one block acts on the first n-qubit state, $|a\rangle$, to calculate $|f(a)\rangle$, and another block acts on $|b\rangle$ to calculate $|f'(b)\rangle$; the second operator is a product operator, in which the product of the last two n-qubit states is calculated, $|c \cdot f'(b)\rangle$. The definition of the latter operator can be found in [27]. In symbols, for the primitive function $f(x)$ the operator $\mathcal{AD}(f)$ looks like:
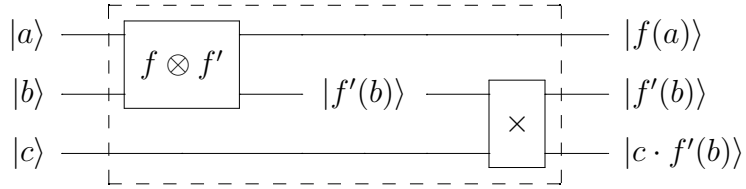
In the quantum circuit above, the $\mathcal{AD}(f)$ operator is indicated by the dashed line.

Once this operator has been defined, the idea of algorithmic differentiation can then be applied as follows. Consider the composite function $g \circ f$. The first iteration of the algorithmic differentiation is to calculate the derivative of f at x_0. It is then possible to calculate the derivative of $g \circ f$ in an iterative way. Let us consider the following input state:

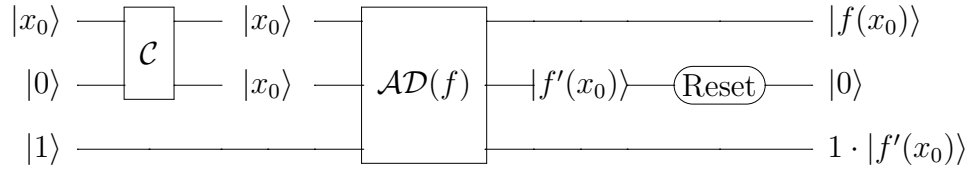$$|s\rangle = |x_0\rangle \otimes |0\rangle \otimes |1\rangle \tag{4}$$

where

- $|0\rangle = |00...0\rangle$,

- $|1\rangle$ is the ket which represents the integer number 1.

In analogy with the tuple used in Section 2, we call this state the *valder state*.

Let us now apply the following algorithm:

- apply the copy operator to the kets $|x_0\rangle$ and $|0\rangle$.

- apply the $\mathcal{AD}(f)$ operator to the first and second kets (both in state $|x_0\rangle$) and to the third ket in state $|1\rangle$.

Equivalently in symbols:

So the output of the first iteration is the state:

$$
\begin{aligned}
|s\rangle &= |f(x_0)\rangle \otimes |0\rangle \otimes |f'(x_0)\rangle & (5)\\
&= |val\rangle \otimes |0\rangle \otimes |der\rangle & (6)
\end{aligned}
$$

where for ease of notation we defined the kets $|val\rangle$ and $|der\rangle$.

Let us now execute the same iteration, in which the only difference is that we use the $\mathcal{AD}(\cdot)$ operator associated to the function g:

- apply the copy operator to the kets $|val\rangle$ and $|0\rangle$.

- apply the $\mathcal{AD}(g)$ operator (in this case associated to g) to the first and second kets (both in state $|val\rangle$) and to the third ket in state $|der\rangle$.

Equivalently in symbols:

8
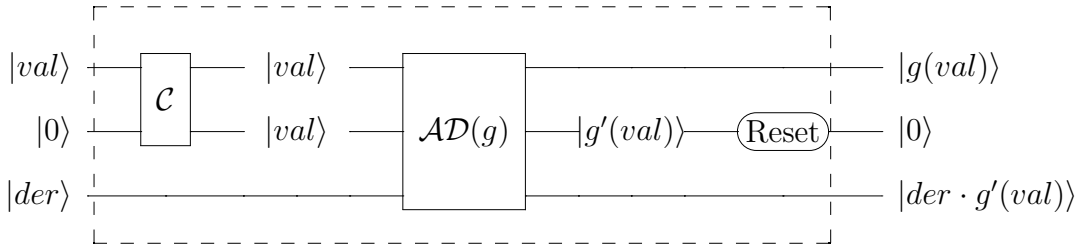
Note that in this way we calculated the derivative of the function $g \circ f$ at x_0.

One can extend this procedure to an arbitrary number of composite functions. To this end, let us define the set of the operators (dashed box) in the previous circuit as the unit B_g, where B stays for block and g refers to the function $g(x)$. The block B_g is represented by the dashed line in the previous circuit.

It is the clear that, if we intend to evaluate the derivative of the arbitrary composite function $f \circ g \circ ... \circ h$, we have to apply the composition of blocks $B_f \times B_g \times ... \times B_h$. The initial input on the left is represented by the $3n$-qubit state $|x_0\rangle\, |0\rangle\, |1\rangle$ and the output on the right is

$$|f(g(...h((x_0)))\rangle\, |0\rangle \left| \left(\frac{df}{dg}\right)_{g_0} \cdot ... \cdot \left(\frac{dh}{dx}\right)_{x_0} \right\rangle . \tag{7}$$

The value in the third n-qubit state is the derivative of the composite function that we were looking for. Note, since the derivative of elementary functions can be calculated on quantum computers, the corresponding boxes can be constructed. Then, the evaluation of the derivative of an arbitrary composite function is a straightforward (and systematic) application of our approach. This is one of the main outcomes of the present paper.

4 Quantum algorithmic differentiation: the framework

In this section we merge the results from the previous two sections to define a quantum algorithmic differentiation ($q\mathcal{AD}$) framework.

Let us first define the functions needed for $q\mathcal{AD}$ in terms of a symbolic class:

```
class QuantumAglorithmicDifferentiator()
    attributes
        'valder' QuantumState |s⟩ = |v⟩ ⊗ |0⟩ ⊗ |d⟩
    methods
        arithmetic operations
        ADplus(|s1⟩, |s2⟩)
            return |v1 + v2⟩ ⊗ |0⟩ ⊗ |d1 + d2⟩
```

```
ADminus(|s⟩)
      return |−v⟩ ⊗ |0⟩ ⊗ |−d⟩
  ADtimes(|s1⟩,|s2⟩)
        return |v1 ∗ v2⟩ ⊗ |0⟩ ⊗ |d1 ∗ v2 + v1 ∗ d2⟩
  ADreciprocal(|s⟩)
        return |1/v⟩ ⊗ |0⟩ ⊗ |−1/v² ∗ d⟩
```
primitive functions
```
  ADexp(|s⟩)
        return |exp(v)⟩ ⊗ |0⟩ ⊗ |exp(v) ∗ d⟩
  ADlog(|s⟩)
        return |log(v)⟩ ⊗ |0⟩ ⊗ |1/v ∗ d⟩
  ADsqrt(|s⟩)
        return |√v⟩ ⊗ |0⟩ ⊗ |0.5/√v ∗ d⟩
  ADsin(|s⟩)
        return |sin(v)⟩ ⊗ |0⟩ ⊗ |cos(v) ∗ d⟩
  ADcos(|s⟩)
        return |cos(v)⟩ ⊗ |0⟩ ⊗ |− sin(v) ∗ d⟩
  ADtan(|s⟩)
        return |tan(v)⟩ ⊗ |0⟩ ⊗ |1/ cos²(v) ∗ d⟩
  ADarcsin(|s⟩)
        return |arcsin(v)⟩ ⊗ |0⟩ ⊗ |1/√(1 − v ∗ v) ∗ d⟩
  ADarctan(|s⟩)
        return |arctan(v)⟩ ⊗ |0⟩ ⊗ |1/(1 + v ∗ v) ∗ d⟩
```

The functions above defined are in fact operators acting on the quantum (*valder*) states $|s⟩ = |v⟩ \otimes |0⟩ \otimes |d⟩$. Note that for sake of simplicity, we assume that the reset procedure $\mathcal{R}$ is always applied at the end of the application of the $\mathcal{AD}(\cdot)$ operators.

In the cases above in which arithmetic operations sum and products are considered, the input consists of two array of three n-qubit states, that is $6n$ qubits in total. The outcome outlined above consists of $3n$ qubits only, which can be identified with the new upper $3n$-qubit state. The lower $3n$-qubit state is irrelevant for the sum of two functions, but may become useful if more functions needed to be summed or multiplied.

If we intend to calculate the derivative of the sum $f(x) + g(x) + h(x)$ at the point x_0, we can consider 3 sets of $3n$-qubits ($9n$ qubits in total). We may then consider 3 initial states $|x_0⟩ |0⟩ |1⟩$ and apply B_f on the first, B_g on the second, and B_h on the third, obtaining $|f(x_0)⟩ |0⟩ |f'(x_0)⟩ , |g(x_0)⟩ |0⟩ |g'(x_0)⟩ ,$

and $|h(x_0)\rangle |0\rangle |h'(x_0)\rangle$. One can then sum the first two of them, getting

$$|f(x_0) + g(x_0)\rangle |0\rangle |f'(x_0) + g'(x_0)\rangle \ , \tag{8}$$

and summing this with the last array, $|h(x_0)\rangle |0\rangle |h'(x_0)\rangle$, which leads to the desired result

$$|f(x_0) + g(x_0) + h(x_0)\rangle |0\rangle |f'(x_0) + g'(x_0) + h'(x_0)\rangle \ . \tag{9}$$

As usual, the required derivative is contained in the third qubit of the previous expression. This procedure, for k functions, requires $3n \cdot k$ qubits (or equivalently $3k$ n-qubit states).

Alternatively, one may reduce the number of n-qubit states to $2 \cdot 3 = 6$ in total. To this end, we start with two copies $|x_0\rangle |0\rangle |1\rangle$, one which goes into the block B_f and one into B_g, obtaining $|f(x_0)\rangle |0\rangle |f'(x_0)\rangle$ and $|g(x_0)\rangle |0\rangle |g'(x_0)\rangle$. We then apply the sum, the upper array being given by Eq. (8). We then assume that it is possible (and convenient) to re-initialize the lower array of the 3 remaining states to $|x_0\rangle |0\rangle |1\rangle$. This can be sent into the block B_h, getting $|h(x_0)\rangle |0\rangle |h'(x_0)\rangle$, which can be summed with the previous outcome, obtaining (in the upper array) the desired result of Eq. (9).

The question if 9 (or in general $3k$) states is faster or slower than 6 states in which a re-initialization of one of the states is applied, is open and depends on future implementations. Probably, for k large enough the second paths becomes favorable. Yet, as a matter of principle, it is possible to perform the sum of k functions by using 6 n-qubit states. Very similar arguments hold also in the case of the calculation of the derivative of the product of k functions.

Finally, once these operators are implemented on a quantum computer, we can compute with arbitrary precision (limited only by the quantum machine precision, i.e. the number of qubits) the derivative of any combination of the functions with the following algorithm:

11

Algorithm $q\mathcal{AD}(|s\rangle, n, m)$

Require: $|s\rangle = |x_0\rangle \otimes |0\rangle \otimes |1\rangle$ is a *valder* state and each sub-state is a n-qubit register with m qubits for its integer part.

1. Determine the components of the computational graph for the Algorithmic Differentiation.

2. Calculate the number of qubits needed.

3. For each component:

    (a) apply the transfer operator $\mathcal{C}$

    (b) apply the component specific $\mathcal{AD}(\cdot)$ operator

    (c) apply the reset procedure on the second n-qubit state

    (d) return the *valder* object

4. **If** arithmetic-operations are involved, for each operation:

    (a) apply the component specific qAD operator

4.1   An example: $f(x) = x \cdot \cos(\log x)$

The computational graph of the function $f(x) = x \cdot \cos(\log x)$ is shown below:

$$s_4 \cdot s_3 \equiv x \cdot \cos(\log x)$$

$$s_4 \equiv x \qquad s_3 \equiv \cos(s_2)$$

$$s_2 \equiv \log(s_1)$$

$$s_1 \equiv x$$
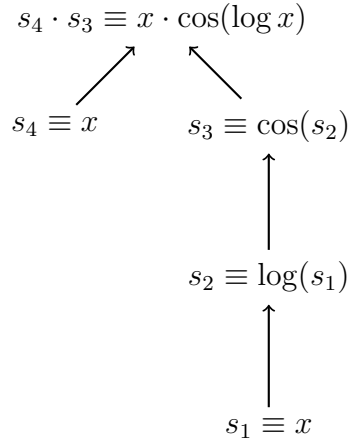
Based on this graph, let us define the quantum states to calculate the derivative of this function.

- $|s_1\rangle = |x_0\rangle \otimes |0\rangle \otimes |1\rangle \equiv |val_1\rangle \otimes |0\rangle \otimes |der_1\rangle$

- $|s_2\rangle = \texttt{ADlog}(|s_1\rangle)$

- $|s_3\rangle = \texttt{ADcos}(|s_2\rangle)$

- $|s_4\rangle = \texttt{ADtimes}(|s_4\rangle, |s_3\rangle)$

Let us now explicitly calculate the *valder* states which will lead to the derivative of the function $f(x) = x \cdot \cos(\log x)$. Using the definition of $\texttt{ADlog}$ we have for $|s_2\rangle$:

$$
\begin{aligned}
|s_2\rangle &= \texttt{ADlog}(|s_1\rangle) \\
&= |\log val_1\rangle \otimes |0\rangle \otimes \left|\frac{1}{val_1} \cdot der_1\right\rangle \\
&= |\log x\rangle \otimes |0\rangle \otimes \left|\frac{1}{x_0} \cdot 1\right\rangle \\
&\equiv |val_2\rangle \otimes |0\rangle \otimes |der_2\rangle
\end{aligned}
$$

For $|s_3\rangle$, using the definition of $\texttt{ADcos}$:

$$
\begin{aligned}
|s_3\rangle &= \texttt{ADcos}(|s_2\rangle) \\
&= |\cos(val_2)\rangle \otimes |0\rangle \otimes |-\sin(val_2) \cdot der_2\rangle \\
&= |\cos(\log x_0)\rangle \otimes |0\rangle \otimes \left|-\sin(\log x_0) \cdot \frac{1}{x_0}\right\rangle \\
&\equiv |val_3\rangle \otimes |0\rangle \otimes |der_3\rangle
\end{aligned}
$$

Finally, using the definition of $\texttt{ADtimes}$ for the states $|s_3\rangle$ and $|s_4\rangle$ we get:

$$
\begin{aligned}
\texttt{qAD}\,[x \cdot \cos(\log x)] &\equiv |f'\rangle = \texttt{ADtimes}(|s_4\rangle, |s_3\rangle) \\
&= |val_4 \cdot val_3\rangle \otimes |0\rangle \otimes |der_4 \cdot val_3 + val_4 \cdot der_3\rangle \\
&= |x \cdot \cos(\log x_0)\rangle \otimes |0\rangle \otimes \left|1 \cdot \cos(\log x_0) + x_0 \cdot \left(-\sin(\log x_0) \cdot \frac{1}{x_0}\right)\right\rangle \\
&= |x_0 \cdot \cos(\log x_0)\rangle \otimes |0\rangle \otimes |\cos(\log x_0) - \sin(\log x_0)\rangle
\end{aligned}
$$

The state $|f'\rangle$ contains then the value of the function and of its derivative, both valuated at point x_0.

13

5 Conclusions

This paper presents a translation of the algorithmic differentiation procedure for a classical computer to a quantum computer and poses the basis for the implementation of a quantum algorithmic differentiation framework for scientific computing.

We first defined the *valder* quantum state which is the input state to calculate the value of a function and its derivative in a given point. The *valder* state is in fact a quantum floating point representation of the point at which we need to calculate the function and its derivative, say x_0. This state is composed by 3 n-qubit states: the first representing the value x_0, the second representing the value 0, and finally the third representing the value 1. The latter two states are needed to calculate the value of the derivative of the function in the quantum algorithmic differentiation framework.

We then defined the operators and procedures necessary to calculate the function and the derivative in x_0. The *transfer* or *copy* operators copies the first n-qubit state onto the second, the *reset* procedure sets the second n-qubit state to represent the value 0 and finally the AD operator is built in order to compute the value of a primitive function and its derivative. We also showed two possible solutions for the reset procedure, one fully quantum and the other applicable in hybrid systems, with the help of a classical bit.

In conclusion, based on the existing definitions of primitive functions on a quantum computer, we outlined the algorithm to compute the value of a composite function and its derivative at a given point, based on the computational graph of the function itself.

We acknowledge the practical complexity of the current implementation of this algorithm due to the large number of qubits needed to represent the value of functions and derivatives in a point, however we do see the potential of implementing this quantum algorithmic differentiation framework into currently available algorithms for scientific computing, optimization and machine learning on quantum computers.

Acknowledgements

References

[1] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120. 1981.

[2] Richard Neidinger. Introduction to automatic differentiation and matlab object-oriented programming. *SIAM Review*, 52:545–563, 01 2010.

[3] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, 2012.

[4] Luca Capriotti. Fast Greeks by Algorithmic Differentiation. *SSRN*, June 2010.

[5] Cristian Homescu. Adjoints and automatic (algorithmic) differentiation in computational finance. *SSRN*, September 2011.

[6] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv e-prints*, page arXiv:1502.05767, February 2015.

[7] Numerical Algorithms Group (NAG). *Exact First- and Second-Order Greeks by Algorithmic Differentiation*. nag.com, 2017.

[8] Marc Henrard. *Algorithmic Differentiation in Finance Explained*. Palgrave Macmillan, Cham, 2017.

[9] Stephen P. Jordan. Fast Quantum Algorithm for Numerical Gradient Estimation. Phys. Rev. Lett., 95(5):050501, July 2005.

[10] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. Evaluating analytic gradients on quantum hardware. Phys. Rev. A, 99(3):032331, March 2019.

[11] Robert M. Parrish, Edward G. Hohenstein, Peter L. McMahon, and Todd J. Martinez. Hybrid Quantum/Classical Derivative Theory: Analytical Gradients and Excited-State Dynamics for the Multistate Contracted Variational Quantum Eigensolver. *arXiv e-prints*, page arXiv:1906.08728, June 2019.

[12] Thomas E. O'Brien, Bruno Senjean, Ramiro Sagastizabal, Xavier Bonet-Monroig, Alicja Dutkiewicz, Francesco Buda, Leonardo DiCarlo, and Lucas Visscher. Calculating energy derivatives for quantum chemistry on a quantum computer. *npj Quantum Information*, 5:113, December 2019.

[13] Dominic W. Berry, Graeme Ahokas, Richard Cleve, and Barry C. Sanders. Efficient Quantum Algorithms for Simulating Sparse Hamiltonians. *Communications in Mathematical Physics*, 270(2):359–371, March 2007.

[14] Patrick Rebentrost, Brajesh Gupt, and Thomas R. Bromley. Quantum computational finance: Monte Carlo pricing of financial derivatives. Phys. Rev. A, 98(2):022321, August 2018.

[15] Ana Martin, Bruno Candelas, Ángel Rodríguez-Rozas, José D. Martín-Guerrero, Xi Chen, Lucas Lamata, Román Orús, Enrique Solano, and Mikel Sanz. Towards Pricing Financial Derivatives with an IBM Quantum Computer. *arXiv e-prints*, page arXiv:1904.05803, April 2019.

[16] Iordanis Kerenidis, Anupam Prakash, and Dániel Szilágyi. Quantum Algorithms for Portfolio Optimization. *arXiv e-prints*, page arXiv:1908.08040, August 2019.

[17] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum algorithms for supervised and unsupervised machine learning. *arXiv e-prints*, page arXiv:1307.0411, July 2013.

[18] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 56(2):172–185, April 2015.

[19] Jeremy Adcock, Euan Allen, Matthew Day, Stefan Frick, Janna Hinchliff, Mack Johnson, Sam Morley-Short, Sam Pallister, Alasdair Price, and Stasja Stanisic. Advances in quantum machine learning. *arXiv e-prints*, page arXiv:1512.02900, December 2015.

[20] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. Nature, 549(7671):195–202, September 2017.

[21] David Beckman, Amalavoyal N. Chari, Srikrishna Devabhaktuni, and John Preskill. Efficient networks for quantum factoring. Phys. Rev. A, 54(2):1034–1063, August 1996.

[22] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Quantum Computation by Adiabatic Evolution. *arXiv e-prints*, pages quant–ph/0001106, January 2000.

[23] J. J. Álvarez-Sánchez, J. V. Álvarez-Bravo, and L. M. Nieto. A quantum architecture for multiplying signed integers. In *Journal of Physics Conference Series*, volume 128 of *Journal of Physics Conference Series*, page 012013, August 2008.

[24] Yudong Cao, Anargyros Papageorgiou, Iasonas Petras, Joseph Traub, and Sabre Kais. Quantum algorithm and circuit design solving the Poisson equation. *New Journal of Physics*, 15(1):013021, January 2013.

[25] Nathan Wiebe and Martin Roetteler. Quantum arithmetic and numerical analysis using Repeat-Until-Success circuits. *arXiv e-prints*, page arXiv:1406.2040, June 2014.

[26] Mihir K. Bhaskar, Stuart Hadfield, Anargyros Papageorgiou, and Iasonas Petras. Quantum Algorithms and Circuits for Scientific Computing. *Quantum Information and Computation*, 2016.

[27] Stuart Hadfield. Quantum Algorithms for Scientific Computing and Approximate Optimization. *arXiv e-prints*, page arXiv:1805.03265, May 2018.

[28] Xiao-Qiang Shao, li Chen, Shou Zhang, and Kyu Yeon. Fast cnot gate via quantum zeno dynamics. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 42:165507, 07 2009.

17