Treball de Fi de Grau

Grau en Enginyeria Física
Grau en Enginyeria en Tecnologies Industrials

Study and development of quantum algorithms for solving and estimating parameters of partial differential equations

Autor: Anna Sallés Rius

Director: Dr. Jordi Tura Brugués

Tutor UPC: Pietro A. Massignan

Convocatòria: 31-08-2022

Resum

Les Equacions Diferencials en derivades Parcials (EDPs) descriuen una gran varietat de fenòmens físics. En moltes situacions, es pot tenir accés a observacions d'un determinat sistema físic i disposar d'alguna idea inicial sobre un aspecte qualitatiu de la seva dinàmica. Aquest coneixement previ és suficient per a determinar l'estructura global de l'EDP, però no els seus coeficients específics. De fet, els paràmetres dels models d'EDPs normalment codifiquen interpretacions científiques rellevants, de manera que és de gran interès poder determinar els seus valors. Aquests coeficients s'estimen a partir de mesures disponibles, que acostumen a presentar soroll. Aquest projecte presenta un algoritme híbrid quàntic-clàssic per a inferir els paràmetres d'una EDP donat un conjunt de dades d'observacions empíriques.

Per tal de dur a terme l'estimació de paràmetres, cal tenir accés a una solució de l'EDP. Aquesta tesi proposa un algoritme quàntic per a resoldre EDPs basat en un circuit quàntic parametritzat. Aquest circuit codifica les variables d'entrada utilitzant una aplicació coneguda com a *Chebyshev feature map* que ofereix una base de polinomis molt representativa i que posseeix gran expressivitat. A continuació, la solució es calcula al circuit quàntic mitjançant mesures de valors esperats. Les derivades espacials i temporals es calculen al circuit quàntic mitjançant derivació automàtica (a través de l'anomenada *parameter shift rule*) de forma analítica, de manera que s'eviten les inexactituds derivades dels procediments que utilitzen diferències finites per a calcular gradients. Per últim, el circuit quàntic s'entrena per tal de satisfer l'EDP donada i les condicions de frontera especificades.

Com a cas d'estudi, l'algoritme s'il·lustra a partir de diverses simulacions per tal de determinar el circuit quàntic que resol l'equació de la calor amb millor expressivitat i exactitud. Amb aquesta configuració es determinen els paràmetres de l'equació de la calor.

Paraules clau: circuits quàntics diferenciables, estimació de paràmetres, equacions diferencials en derivades parcials, algoritmes quàntics, computació quàntica, aprenentatge automàtic quàntic

Resumen

Las Ecuaciones Diferenciales en derivadas Parciales (EDPs) describen una gran variedad de fenómenos físicos. En muchas situaciones, se puede tener acceso a observaciones de un determinado sistema físico y disponer de alguna idea inicial sobre un aspecto cualitativo de su dinámica. Este conocimiento previo es suficiente para determinar la estructura global de la EDP, pero no sus coeficientes específicos. De hecho, los parámetros de los modelos de EDPs normalmente codifican interpretaciones científicas relevantes, de manera que es de gran interés poder determinar sus valores. Estos coeficientes se estiman a partir de medidas disponibles, que acostumbran a presentar ruido. Este proyecto presenta un algoritmo híbrido cuántico-clásico para inferir los parámetros de una EDP dado un conjunto de datos de observaciones empíricas.

Para la estimación de parámetros, hace falta tener acceso a una solución de la EDP. Esta tesis propone un algoritmo cuántico para resolver EDPs basado en un circuito cuántico parametrizado. Este circuito codifica las variables de entrada usando una aplicación conocida como *Chebyshev feature map* que ofrece una base de polinomios muy representativa y que posee gran expresividad. A continuación, la solución se calcula en el circuito cuántico mediante medidas de valores esperados. Las derivadas espaciales y temporales se calculan en el circuito cuántico mediante diferenciación automática (a través de la *parameter shift rule*) de forma analítica, evitando así las inexactitudes derivadas de los procedimientos que usan diferencias finitas para calcular gradientes. Por último, el circuito cuántico se entrena para satisfacer la EDP dada y las condiciones de frontera especificadas.

Como caso de estudio, el algoritmo se ilustra a partir de varias simulaciones con el fin de determinar el circuito cuántico que resuelve la ecuación del calor con mejor expresividad y exactitud. Con esta configuración se determinan los parámetros de la ecuación del calor.

Palabras clave: circuitos cuánticos diferenciables, estimación de parámetros, ecuaciones diferenciales en derivadas parciales, algoritmos cuánticos, computación cuántica, aprendizaje automático cuántico

Abstract

Partial differential equations (PDEs) describe a wide variety of physical phenomena. In many situations, one can have access to observations on some physical system and some initial idea of some qualitative aspects of its dynamics. This prior knowledge is enough to determine the overall structure of the PDE, but not its specific coefficients. In fact, the parameters of PDE models encode insightful scientific interpretations, so it is of great interest to determine their values. These coefficients are estimated from the available noisy measurements of the system. This project presents a hybrid quantum-classical approach to infer the parameters of a PDE given a data-set of empirical observations.

In order to perform parameter estimation, it is necessary to have access to a PDE solver. This thesis proposes a quantum algorithm to solve PDEs based on a parameterized quantum circuit. This circuit encodes the input variables in a Chebyshev quantum feature map that offers a powerful basis set of fitting polynomials and possesses rich expressivity. Then, the surrogate of the real solution is computed by measuring expectation values. The spatial and temporal derivatives of the surrogates are computed in the differentiable quantum circuit (DQC) through automatic differentiation (via the so-called parameter shift rule) in an analytical form, thus avoiding inaccurate finite difference procedures for calculating gradients. The DQC is then trained to satisfy the given PDE and specified boundary conditions.

As a case study, the algorithm is illustrated via several simulations in order to determine the DQC that solves the Heat equation with best expressivity and accuracy. The parameters of the Heat equation are then estimated with this particular setting.

Keywords: Differentiable Quantum Circuits, Parameter inference, Partial Differential Equations, Quantum Algorithms, Quantum Computing, Quantum Machine Learning

Contents

List of Figures

List of Tables

List of Acronyms

The acronyms used in this thesis are summed up in Table 1.

Acronym	Full description
AD	Automatic Differentiation
DQC	Differentiable Quantum Circuit
IBVP	Initial Boundary Value Problem
HHL	Harrow-Hassidim-Lloyd algorithm
LCU	Linear Combination of Unitaries
MAE	Mean Absolute Error
MAX	Maximum Absolute Error
MSE	Mean Square Error
NISQ	Noisy Intermediate-Scale Quantum
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
PINN	Physics-Informed Neural Network
PQC	Parameterized Quantum Circuit
QLSA	Quantum Linear System Algorithm
QML	Quantum Machine Learning
QNN	Quantum Neural Network
SGD	Stochastic Gradient Descent
UFA	Universal Function Approximator
VQE	Variational Quantum Eigensolver
VQA	Variational Quantum Algorithm

Table 1: Acronyms used in the thesis

1 Introduction

It has recently been discovered that certain properties of quantum mechanics can be applied to computation in what is known as quantum computer, a machine designed to process information based on the laws of quantum mechanics. More specifically, by using the superposition principle and the interference[1] effect of quantum mechanics [4] [5], quantum computers appear to be more powerful than the standard classical Turing machine model of theoretical computer science. Quantum computing is the filed that investigates the computational power and other properties of computers based on quantum-mechanical principles. The multiple applications of quantum computing rely on quantum algorithms, which are algorithms that run on a quantum computer and achieve a speedup over some classical algorithm solving the same problem. In fact, using only polynomial resources, these quantum algorithms can compute certain functions which are not known to be computable on classical digital computers in less than exponential time [6]. Some of the main reasons that justify the power of quantum computing are the following [7]:

- Quantum algorithms can be used for some classically intractable problems. There are some problems that are hard for classical computers, but are easy to solve with quantum algorithms[2]. The difficulty of a problem is classified into complexity classes, a key concept in computational complexity theory [8]. If a problem is said to be complete for a complexity class, it is essentially one of the 'hardest' problems in that class and every other problem within that class essentially reduces to it.

- Quantum algorithms achieve an exponential speedup over some classical methods. The speedup achieved thanks to quantum computers is normally measured according to computational complexity theory. Hence, the efficiency of a quantum algorithm is measured in terms of the complexity of the algorithm. In both classical and quantum settings, the runtime is measured by the number of elementary operations used by an algorithm. In the case of quantum computation, this can be measured using the quantum circuit model. A quantum circuit is a sequence of elementary quantum operations called quantum gates, each applied to a small number of qubits (quantum bits). One measures the "asymptotic" complexity of the algorithm as the size of the input grows. The performance of algorithms is compared using the notation $\mathcal{O}\left(f(n)\right)$, which can be understood as "asymptotically" upper-bounded by $f(n)$ [9].

- No known classical algorithm can simulate a quantum computer efficiently in its full generality.

Since the use of the properties of quantum mechanics allows to speed up certain computations, the interest in quantum computation has been recently growing. However, quantum computing presents several difficulties [10]:

- *Decoherence*[3] tends to destroy the information in a superposition of states in a quantum

[1] Phenomenon that occurs when the positive and negative amplitudes for a quantum state cancel each other.

[2] The best known example is the problem of finding the prime factors of a large integer [6].

[3] Quantum computation involves manipulating the quantum states of objects that are in coherent quantum superpositions. These superpositions, however, tend to be quite fragile and decay easily; this decay phenomenon is called decoherence. One way of thinking about decoherence is to consider the environment to be "measuring" the

computer, thus making long computations impossible.

- *Inaccuracies*[4] in quantum state transformations can accumulate over the course of a computation, rendering the output of long computations unreliable.

In fact, it is very challenging to build a quantum system that satisfies all of these criteria. Some **quantum error correcting** codes[5] have been built that are able to reduce both decoherence and inaccuracy during transmission and storage of quantum data [11]. The idea of quantum error correction is to encode the quantum system in a very highly entangled state in order to protect it from damage. Unfortunately, there is a significant cost for doing quantum error correction, because writing the protected quantum information into a highly entangled state requires many additional qubits, so this discovery is not sufficient to ensure that a noisy quantum computer can perform reliably. Hence, reliable quantum computers using quantum error correction are not likely to be available very soon [7].

A device that works effectively even when its elementary components are imperfect is said to be **fault-tolerant**. Through application of quantum error correction schemes, a fault-tolerant quantum computer is able to avoid the uncontrollable cascade of errors caused by the interaction of qubits.

Even though fault-tolerant quantum computers will likely not be available in the near future, in 2016 access was granted to the first cloud-based quantum computer. The actual quantum processors have been called Noisy Intermediate-Scale Quantum (NISQ)[6] computers [7].

1.1 Quantum Computing applied to PDEs

One of the fields in which quantum algorithms can be applied is solving large systems of linear equations. The best known quantum algorithm for solving systems of linear equations is the one developed by Harrow, Hassidim and Lloyd (HHL) [12]. This same algorithm, or closely related ideas, can also be applied to problems related to linear equations such as solving differential equations, data fitting and various tasks in machine learning.

In fact, quantum computers possess algorithmically superior scaling for certain problems that include amplitude amplification and Abelian hidden subgroup problems [8] [5]. For linear algebra tasks, quantum computers have an exponential advantage over classical computers when

state of a quantum system by interacting with it. The laws of quantum mechanics establish that a quantum system cannot be observed without producing an uncontrollable disturbance in the system. So in order to use a quantum system to store and reliably process information, the system needs to be kept perfectly isolated from the outside world. Decoherence prevents this isolation.

[4] As with an analog classical computer, the state of a quantum superposition depends on certain continuous parameters. For instance, one of the most common quantum gates used in quantum computations is a rotation of a quantum bit by an angle θ. When applying this transformation, there will be some inherent inaccuracy in this angle θ.

[5] To carry out a quantum error-correction protocol, the quantum information that needs to be protected must first be encoded. Then some recovery operations are repeatedly performed in order to reverse the errors that accumulate. Since encoding and recovery are themselves complex quantum computations, errors will inevitably occur while performing these operations. Thus, the methods for recovering from errors need to be robust enough so as to succeed with high reliability even when committing some errors during the recovery step [10].

[6] "Intermediate Scale" refers to the size of quantum computers (with a number of qubits ranging from 50 to a few hundred). "Noisy" emphasizes the imperfect control over those qubits, since the noise places serious limitations on what quantum devices can achieve.

it comes to solving linear system of equations (as offered by the HHL algorithm [12]), since they give a very significant speed up [13].

Differential equations (DEs), which include ordinary differential equations (ODEs) and partial differential equations (PDEs), are used to model a wide variety of physical phenomena. ODEs describe the dynamics of continuously changing processes by relating a process and its rate of change. PDE models are commonly used to model complex dynamic system in applied science such as physics, biology and finance. Annex 9.1 introduces some mathematical notation and theoretical background of PDEs that may help the reader follow this thesis.

PDEs are the expression of processes that occur across time and space. Most scientific phenomena can typically be modelled by the evolution of a given quantity through space and time, and thus can be translated into a PDE. Hence PDEs describe many engineering phenomena and processes occurring in nature, and their field of application is very wide. Here are some examples: plasma physics, finance, biology, wave propagation, fluid mechanics (air or liquid), vibration, mechanics of solids, heat flow, electric fields and potentials, diffusion of chemicals in air or water, electromagnetism and quantum mechanics...[14]

Although PDEs are extremely hard to solve (in general), classical computers are already able to perform these calculations. In classical computers, PDEs are often solved by discretization in order to produce a system of linear equations rather than a system of one or more differential equations, since a system of discrete linear equations is indeed much easier to solve. Therefore, being able to solve PDEs more efficiently than all known classical algorithms, using a quantum computer, could significantly accelerate scientific progress. These developments, amongst others, rely on the nascent field of quantum machine learning (QML) [15] .

1.2 Objective and research question

In many situations, an observed process can be modelled in the form of a differential equation. While its solving constitutes an important task in phenomenological understanding, in many practical settings this equation is not fully known. In such cases, one may have access to observations on some target system and some initial idea of some qualitative aspects of its dynamics (e.g. Hamiltonian learning). This prior knowledge and understanding of the studied dynamic system leads to the proposal of some model for a PDE. Even if the overall structure of the PDE is known, its specific coefficients might often be unknown and need to be estimated from noisy measurements of the system. In fact, the parameters of PDE models encode insightful scientific interpretations, so it is of great interest to determine their values by inferring them from empirical observations.

A lot of statistical methods have been developed to estimate the parameters in PDE models by repeatedly solving the models numerically, which is time consuming and leads to a high computational load. Therefore, there is a growing need to develop efficient estimation methods for PDE models. And this is the problem that will be addressed in this thesis. This project targets parameter inference in partial differential equations, which is a direct application of scientific machine learning and modelling in general that combines theoretical models with empirical observations.

Hence, the objective of this project is to provide a framework to estimate the coefficients of the PDE that models some given dataset in a quantum approach, by implementing a parameterized quantum circuit.

1.3 Structure of the project

This thesis has been divided into several sections in order to facilitate its understanding. Section 2 takes a look into the theoretical background of the different existing approaches for solving PDEs, from classical methods (section 2.1) to quantum strategies (section 2.2). Section 3 gives the general approach that will be followed through this project.

The main part of the thesis is then structured into two big sections: the theoretical description and analysis of the parameterized quantum circuit and the optimization procedure (section 4) and its practical implementation (section 5).

On one hand, the theoretical approach consists on: (1) establishing a solver for the PDE (section 4.1), (2) describing and building the structure of the established solver (section 4.2), (3) defining the optimization strategy and training the circuit parameters that allow to reproduce the solution function (section 4.3).

On the other hand, the practical section contains a description of the experimental results obtained through the built quantum circuit. Section 5.2 discusses which quantum circuit enables the best expressivity, and section 5.3 analyses the parameter inferring technique for the most expressive quantum circuit.

Finally, section 6 offers a discussion of the results and dives into the limitations of the work done. Section 7 explores some open questions and improvements, and takes a quick look at potential applications of the work presented in this thesis.

2 State of the art: PDEs solvers

In the recent years, a lot of work has been done related to solving PDEs with numerical strategies, ranging from classical to quantum methods. This section will analyse some of these solving techniques and the theoretical background behind each type of solver.

What these solvers have in common is the need to calculate the partial derivatives of the objective function u that participates in the differential equation. Each one of them follows a different approach.

2.1 Classical methods

Classical numerical methods have complexity that grows exponentially in the dimension, a phenomenon called *curse of dimensionality* that constitutes a major challenge. The most common approaches to solving PDEs on a digital computer are classified into local methods and global methods.

2.1.1 Local methods

Local approaches are also known as **grid or mesh methods**, since they rely on the discretization of the space of variables. In this case, derivatives are approximated using numerical differentiation techniques (such as finite differencing [16] and Runge-Kutta methods).

The definition of the derivative is given by:

$$\frac{du}{dx} = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h}.$$

Since data is always sampled at a finite spacing, in practice a derivative must always be approximated:

$$\frac{du}{dx} \approx \frac{u(x+h) - u(x)}{h}.$$

This equation is the basis of numerical differentiation, where data is sampled on a grid with spacing Δx, allowing the use of a more natural notation in terms of the grid index i. The derivative can then be approximated as:

$$\frac{du_i}{dx} \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x}.$$

The finite differences method (FDM) is based on discretizing space into a regular lattice, solving a system of linear equations that approximates the PDE on the lattice, and output the solution on those grid points. If each spatial coordinate has n discrete values, then n^d points are needed to discretize a d-dimensional problem. The exponential scaling in the dimension is a great difficulty of classical methods.

Numerical differentiation through finite differences (FD) has some important drawbacks [17]:

- The precision of the derivatives depends on the step Δx, hence the approximation is only valid for small Δx. If the samples are spaced far apart, the resulting derivative will be inaccurate. And if there are abrupt changes between the samples (for example, in a delta function), the derivative at those points will be inaccurate.

- Higher order derivatives are calculated by successively applying the approximation. Consequently, small errors in lower orders get propagated and amplified to higher orders, making estimates of these higher-order derivatives highly inaccurate (see equation 1 as an example for a second order derivative).

$$\frac{d^2u}{dx^2} \approx \frac{\frac{u(x+h)-u(x)}{h} - \frac{u(x)-u(x-h)}{h}}{h} = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \tag{1}$$

- At the edges of the dataset (boundary terms), the derivatives cannot be calculated. Doing so would require extrapolating, which is highly inaccurate, so typically the data at the edge is simply discarded.

- Since this method carries important errors in numerical differentiation, which lead to inaccurate derivatives, it is not suitable on noisy experimental data.

Another mesh-method is the finite element method (FEM), which is based on approximating the solution on an assemblage of simply shaped (triangular, quadrilateral) finite pieces or "elements" which together make up complexly shaped domains. Therefore, this method is suitable for irregular and complex geometries, and it is mainly used when the boundary conditions (BC) are irregular or very complicated, but not for ordinary or simple BC. In this method, the differential equation is multiplied by functions with local support (restricted by the grid) and then integrated. This produces a set of equations that the solution must satisfy, which are then used to approximate the solution.

Another grid-based method is the finite volume method (FVM), which considers a grid by dividing space into volumes or cells. The equation is then integrated over the volume. Applying the divergence theorem, the volume integral over the cells is converted into a surface integral across the boundaries of the cells. This method evaluates expressions for the average value of the solution over some volume and uses this data to construct approximations of the solution within the cells.

2.1.2 Global methods

An alternative to grid methods are global methods, also known as **spectral methods**. Spectral methods use linear combinations of basis functions (such as Fourier basis states or Chebyshev polynomials) to globally approximate the solution. These basis functions allow the construction of a linear system whose solution approximates the solution of the PDE.

Global methods work with surrogates[7] $p(x)$, which need to have enough degrees of freedom to approximate the data $u(x)$ reasonably: $p(x) \approx u(x)$.

A classical approach is to represent $p(x)$ as a series expansion, in terms of a suitable basis set $h(x)$.

$$p(x) = \sum_n a_n h_n(x).$$

[7] Surrogates (also known as digital twins) are a widely used approach to approximate a dataset by a different data-driven model with certain desirable properties [17].

This recasts the problem to finding the optimal coefficients a_n for the polynomial approximation of the sought function, such that $p(x) \approx u(x)$. The derivatives are then easily and accurately calculated, since it is simply the sum over the derivatives of the basis functions:

$$\frac{du}{dx} \approx \frac{dp(x)}{dx} = \sum_i a_i \frac{dh_i(x)}{dx}.$$

The flexibility of the surrogate strongly depends on the choice of basis functions. The most well-known choices that form a basis are the following:

- Polynomials: $h_k(x) = x^k$

  The main limitation of polynomials is that they do not have enough expressivity to model most data. However, data can locally be approximated using a spline interpolation, a method that locally fits a polynomial in a sliding window and ensures continuity at the edges. They can also be used to smooth data, so they are a widely-used choice to calculate derivatives. However, they offer inaccurate results with noisy data.

- Fourier basis: $h_k(x) = e^{2\pi i k x}$

  Using Fourier series has several properties: they are computationally efficient, well established, and are particularly useful for calculating derivatives. Furthermore, the Fourier representation allows to denoise the data by applying a low-pass filter. This is also known as the spectral method to calculate derivatives, and it is often used when solving PDEs. However, this method is not suitable when treating with nonlinear terms, since they correspond to convolutions in Fourier spaces, making the derivative calculation much more complicated.

- Chebyshev basis

  The Chebyshev polynomials are two sequences of polynomials related to the cosine and sine functions, notated as $T_k(x)$ and $U_k(x)$. The Chebyshev polynomials of the first kind $T_k(x)$ are defined by $T_k(\cos\theta) = \cos k\theta$. Similarly, the Chebyshev polynomials of the second kind $U_k(x)$ are defined by $U_k(\cos\theta)\sin\theta = \sin((k+1)\theta)$. These expressions define polynomials in $\cos\theta$, hence they are bounded in the interval $[-1, 1]$.

  The Chebyshev polynomials of the first kind $\{T_k(x)\}_{k\geq 0}$ are defined recursively by:

  $$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x), \;\; T_0(x) = 1, \;\; T_1(x) = x, \;\; \forall k \geq 1.$$

  The Chebyshev polynomials of the second kind $\{U_k(x)\}_{k\geq 0}$ are defined recursively by:

  $$U_{k+1}(x) = 2xU_k(x) - U_{k-1}(x), \;\; U_0(x) = 1, \;\; U_1(x) = 2x, \;\; \forall k \geq 1.$$

  The crucial properties of Chebyshev polynomials are their chaining properties, nesting properties and simple differentiation rules, which account for the power representation of these polynomials in approximation theory. In fact, the set of Chebyshev polynomials form an orthonormal basis, so that any smooth function $f(x)$ can be represented via the expansion $f(x) = \sum_{k=0}^{\infty} a_k T_k(x)$ on $-1 \leq x \leq 1$, so they offer a very accurate fitting for oscillating functions defined in the region $x \in (-1, 1)$.

Since a Chebyshev series is related to a Fourier cosine series through a change of variables, all of the theorems, identities and characteristics that apply to Fourier series have a Chebyshev counterpart. In fact, Chebyshev polynomials are the most widely-used basis functions used in spectral methods, and they are commonly used in spectral algorithms for solving PDEs [3].

2.1.3 Physics Informed Neural Networks (PINNs)

On classical hardware, a neural network consists of a matrix multiplication composed with a non-linearity function,

$$h(x) = f(xW^T + b)$$

where x is the input, W the kernel or weight matrix, b the bias and f the non-linearity function [17]. These layers are composed to increase expressive power, yielding a deep neural network,

$$g_\theta(x) = h_n \circ h_{n-1} ... \circ h_0(x)$$

where θ are the network's weights and biases.

Neural networks are *universal function approximators* (UFAs) [17], meaning that a network with a single hidden layer of infinite width can approximate any continuous function. They are used to represent an approximate surrogate solution $\hat{u}$, since they scale well to higher dimensions, are computationally efficient and are very flexible.

Physics Informed Neural Networks (PINNs) are one of the most widely-used ways to include physical knowledge in neural networks and to obtain differentiable surrogates that respect this physical prior information, therefore they have a lot of applications [18]. PINNs use (higher-order) derivatives of neural network outputs with respect to neural network inputs as terms to train neural networks [19]. Consequently, PINNs have become a very widely-used method for both solving PDEs (given an equation, boundary conditions, and some measurements) and for parameter inference, for various reasons [17]:

- They do not require specialized architectures or advanced numerical approaches, so they are very straightforward to implement. They are not mesh methods, since the data or solution are parameterized as a neural network. Not having to create meshes is a simplifying property.
- Automatic differentiation can be used to calculate the derivatives, yielding machine-precision derivatives.
- When used for parameter inference, PINNs essentially act as consistent denoisers, making them particularly robust and useful when working with noisy and sparse data.

2.2 Quantum methods

The local and global classical algorithms described above often consider the problem of outputting the solution at N points in space, which requires $\Omega(N)$ space and time. Quantum algorithms often (though not always) consider the alternative problem of outputting a quantum state proportional to such a vector, which requires only $\Omega(\log N)$ space. The fact of working with quantum states provides more limited access to the solution, but it can potentially be done in only $poly(\log N)$ time [20].

2.2.1 Discretization

In the recent years, a growing number of quantum algorithms have been developed for solving PDEs. These quantum PDE solvers use several strategies, most of which are based on discretization: either discretizing space or frequency domain (hence working in a Fourier or Chebyshev subspace).

The common steps to solve a partial differential equation with a quantum computer are the following:

1. **Discretization of the PDE**: Let $f(\mathbf{x}, t)$ be a function which is a solution of a given partial differential equation, where $\mathbf{x}$ is a d-dimensional vector. In order to store and manipulate the solution of the PDE on a quantum device, discretization is applied either in space (analogous to classical grid-based methods) or in frequency (analogous to classical spectral methods).

2. **Mapping**: Most quantum algorithms apply either Hamiltonian simulation [21] or the Quantum Linear System Algorithm (QLSA) [12].

   (a) Schrödinger's equation: the PDE is transformed into Schrödinger's equation, and the problem is solved with Hamiltonian simulation techniques.

   The time evolution of the state of a closed quantum system is described by the Schrödinger equation [5]

   $$i\hbar \frac{d\,|\psi\rangle}{dt} = H\,|\psi\rangle\,. \tag{2}$$

   It is common to absorb the factor $\hbar$[8] into H, effectively setting $\hbar = 1$. H is a fixed Hermitian operator known as the Hamiltonian of the closed system.

   The basis of simulation is the solution of differential equations which capture the physical laws governing the dynamical behavior of a system. Many simple quantum systems are governed by Schrödinger's equation 2. Then, an approach to solve PDEs is to map the equation to a Hamiltonian simulation problem. For Hamiltonian simulation, the matrix describing the system A must be turned into an Hermitian matrix H. Then, the goal of Hamiltonian simulation is to solve Schrödinger equation

   $$\frac{d\,|\psi\rangle}{dt} = iH\,|\psi\rangle$$

   where H is a Hermitian $N \times N$ matrix, by implementing the unitary evolution

   $$|\psi\rangle = e^{-iHt}\,|\psi_0\rangle$$

   using a universal gate set. The problem of Hamiltonian simulation consists on implementing e^{-iHt} as a quantum circuit of gates. An efficient approximation of the solution to this problem is possible for many classes of Hamiltonian, since in most physical systems, the Hamiltonian can be written as a sum over many local interactions [21].

   $$H = \sum_{k}^{n} H_k.$$

[8] Physical constant given by Planck's constant (h) divided by 2π

Trotter formula states that, given A and B some Hermitian operators, then [5]:

$$e^{i(A+B)t} = \lim_{n\to\infty} \left(e^{iAt/n} e^{iBt/n}\right)^n .$$

Then one can apply Trotterization in order to obtain a decomposition for short-time simulations [5]:

$$e^{-iHt} \approx e^{-iH_1 t} \cdots e^{-iH_n t}.$$

Hence, each unitary $e^{-iH_k t}$ can be efficiently implemented.

Hamiltonian simulation can also be solved by writing the Hamiltonian as a linear combination of unitaries (LCU) and applying the so-called LCU lemma [4].

(b) Linear system: solving the PDE reduces to solving a linear system of equations using QLSAs.

For a linear system $A\mathbf{x} = \mathbf{b}$, a QLSA outputs a quantum state proportional to the solution **x**. To learn information about the solution **x**, the output of the QLSA must be post-processed.

The Harrow-Hassidim-Lloyd (HHL) algorithm is used for solving large systems of linear equations [12]. Since most classical algorithms for PDEs involve matrix inversion, the HHL algorithm is useful as a mapping method for solving PDEs. Given an $n \times n$ real matrix A, and a vector **b**, the HHL algorithm can solve the systems of linear equations given by $A\mathbf{x} = \mathbf{b}$ in an amount of time that scales only logarithmically with n, the number of equations and unknowns. Classically, n^2 steps are required to examine all of the entries of A, and n steps are needed to write down the solution vector **x**. In contrast, by taking advantage of the exponential character of the wave function, HHL solves the system of n equations in $log(n)$ steps, and the solution is stored in the amplitudes of a quantum state

$$|\mathbf{x}\rangle = \sum_i x_i |i\rangle .$$

The power of HHL-based algorithms relies then on the amplitude encoding, since this allows to compress large grids into a small qubit register, providing exponential memory advantage. However, there are several caveats and drawbacks that should be considered [22]:

i. The output of the HHL algorithm is not **x** itself, but rather a quantum state $|x\rangle$ of $\log_2 n$ qubits which approximately encodes the entries of **x** in its amplitudes. This 'approximately' means that the algorithm obtains a functional of the solution with an error less than ϵ. Even in this quantum approach, reading out the solution $\mathbf{x} = (x_1, ..., x_n)$ requires exponential sampling, representing a so-called data "output problem". Having access to these amplitudes (hence to extract the value of any specific entry x_i) implies measuring the outcome through quantum state tomography, losing the exponential speed-up. Then, the HHL algorithm allows to extract information of the solution - a functional of the solution - but not the solution itself. Therefore, the HHL algorithm is only useful when used as a primitive for other algorithms, or when only a functional of the solution is needed.

ii. The HHL algorithm requires the preparation of the input state $|\mathbf{b}\rangle = \sum_{i=1}^{n} b_i |i\rangle$. General states cannot be prepared efficiently on a quantum computer. This preparation requires sophisticated techniques such as quantum random access memory (QRAM), thus leading to a so-called data "input problem". Classical data has to be loaded into QRAM, and this cost kills the exponential speedup.

iii. The matrix A should be sparse[9] in order for the quantum computer to efficiently apply the unitary transformation e^{-iAt}.

iv. The matrix A needs to be invertible or 'well-conditioned' so that its condition number κ[10] is bounded.

The complexity of the HHL algorithm has been improved further to

$$\mathcal{O}\left(\kappa s \log(n) \log(1/\epsilon)\right)$$

with κ the condition number, s the sparsity and ϵ the precision [23].

3. **Measurement**: The solution is now stored as a quantum state. The last step is to perform measurements to obtain a functional of the solution, since the global solution cannot be easily extracted. There are many functions of a state that can be extracted efficiently, such as the inner product with another state or a small set of amplitudes.

Using this methodology, combined with other methods, several PDEs have already been solved using quantum algorithms. Several studies have developed generic quantum solvers for differential equations, and have shown algorithmic improvements in terms of accessing quantum oracles[11]. For instance: Poisson equation [24] (generalized to elliptic equations [20]), heat equation [25], wave equation [26]. Not only have second order linear PDEs been solved, but also stochastic PDEs (which arise in mathematical finance) [27] [28] and nonlinear PDEs (such as Burger's equation) [29].

2.2.2 Variational Quantum Algorithms

Variational Quantum Algorithms (VQAs) have emerged as the leading strategy to obtain quantum advantage on NISQ devices, due to their optimization-based or learning-based approach (which consists on using a classical optimizer to train a parameterized quantum circuit). This approach allows to overcome the contraints imposed by NISQ computers (limited number of qubits, limited connectivity of the qubits, and coherent and incoherent errors that limit quantum circuit depth).

In particular, NISQ devices may provide advantages over classical methods when it comes to using an hybrid quantum-classical approach. This hybrid algorithm corresponds to the varia-

[9] A $n \times n$ matrix is said to be s-sparse if it contains at most s nonzero entries per row, for some $s << n$

[10] The condition number κ is defined as the ratio in magnitude between the largest and smallest eigenvalues of a matrix A $\left(\kappa = |\frac{\lambda_{max}}{\lambda_{min}}|\right)$. This number is important for numerical stability issues in solving systems of linear equations

[11] A quantum oracle is a "black box" operation that is used in quantum algorithms for the estimation of a function $f(\mathbf{x})$ given an input $\mathbf{x}$ and using qubits. The oracle is the black box implementation of this function $f(\mathbf{x})$, that is later used as input to another algorithm. The oracle takes a n-bit binary input $\mathbf{x} = (x_0, ...x_{n-1})$ and produces a m-bit binary output $f(\mathbf{x})$, such that $f : \{0,1\}^n \mapsto \{0,1\}^m$.

tional quantum eigensolver (VQE), where a quantum computer is used to prepare parameterized quantum states and classical optimization is used for finding the best variational parameters.

The rise of variational quantum eigensolvers (VQE) has launched further development of variational quantum algorithms (VQAs). Since both VQAs and classical neural networks can be thought of as layers of connected computational units controlled by adjustable parameters, VQAs are also referred to as quantum neural networks (QNNs) [1].

VQAs are performed by parameterized quantum circuits (PQCs), which are typically composed of fixed gates (e.g. controlled NOTs) and adjustable gates (e.g. qubit rotations). Even at low circuit depth, some classes of PQCs are capable of generating highly non-trivial outputs, hence allowing a remarkable expressive power. As an illustrative example, it was proven that constant-depth quantum circuits are more powerful than constant-depth classical circuits, and that the quantum advantage that comes from the quantum correlations present in quantum circuits cannot be reproduced in analogous classical circuits [30]. Also, the power of VQAs for quantum machine learning tasks (training models to fit datasets and making predictions) comes from the high expressivity of quantum circuits, where data points are mapped to quantum states.

Quantum computers can be used to construct solutions to differential equations via variational procedures using a Differentiable Quantum Circuit (or Derivative Quantum Circuit - DQC). A DQC is a type of PQC that implements a quantum neural network that can be analytically differentiated with respect to variational parameters and with respect to the network inputs. DQCs deal with functions and their derivatives using automatic differentiation rules, hence allowing to solve differential equations in a variational approach.

The recent progress in automatic differentiation on quantum computers has allowed to generalize classical scientific machine learning procedure to a quantum setting. These techniques have already led to quantum methods for solving differential equations [3] which are analogous to methods involving classical neural networks [31].

Although VQAs still face important challenges[12], they have recently been proposed for essentially all practical applications of quantum computers. The specific applications where VQAs can provide quantum advantage include chemistry and material sciences, nuclear and particle physics, data analysis, optimization and machine learning [2].

[12]The main challenges include trainability, accuracy and efficiency

3 Problem approach

The general approach consists on inferring coefficients of known PDEs from artificial data. Initially, there is some available data that is governed by a PDE. The form of the PDE is known, but the parameters are unknown. The goal is to infer the coefficients of this PDE; and this discovery process will be performed using a differentiable quantum circuit.

Since the purpose of this thesis is to build a quantum circuit that is able to represent an accurate solution of a given PDE rather than solving the PDE, the chosen PDE does not have to be very complex. To this end, the PDE that has been chosen as an example to train the quantum circuit is the one-dimensional Heat equation, which is the prototypical example of a parabolic PDE. Additionally, it is one of the most widely studied topics in pure mathematics, and its analysis is regarded as fundamental to the broader field of partial differential equations. In fact, it has been proved that quantum computers can achieve a polynomial speedup over classical algorithms for solving the Heat equation [13].

The optimization problem will be treated using a hybrid approach of quantum and classical hardware to find approximate solutions. The hybrid computation includes classical and quantum subroutines, executed on different devices. By implementing some subroutines on classical hardware, the requirement of quantum resources is significantly reduced, particularly the number of qubits, circuit depth and coherence time. In fact, these hybrid algorithms turn out to be successful when used for machine learning problems [1]. The general hybrid approach is made of three main components, as shown in figure 1: the human, the classical computer and the quantum computer. The role of the human is to set up the model using prior information and assess the learning process (although in some cases the circuit can also be found via reinforcement learning). Within the hybrid system, the quantum computer prepares quantum states according to a set of parameters and performs measurements. Measurement outcomes are post-processed by the classical computer, that implements a learning algorithm that adjusts the parameters in order to minimize an objective function. The updated parameters, now defining a new quantum circuit, are fed back to the quantum hardware in a closed loop, as shown in figure 2.

This project uses a hybrid quantum-classical approach, since the solver is a quantum neural network and the optimization process for finding the best variational parameters of the quantum circuit is done classically. The gradients w.r.t. inputs (in this example t and x) will be estimated from the quantum circuit, and the gradients w.r.t parameters θ and the function coefficients w will be computed classically.

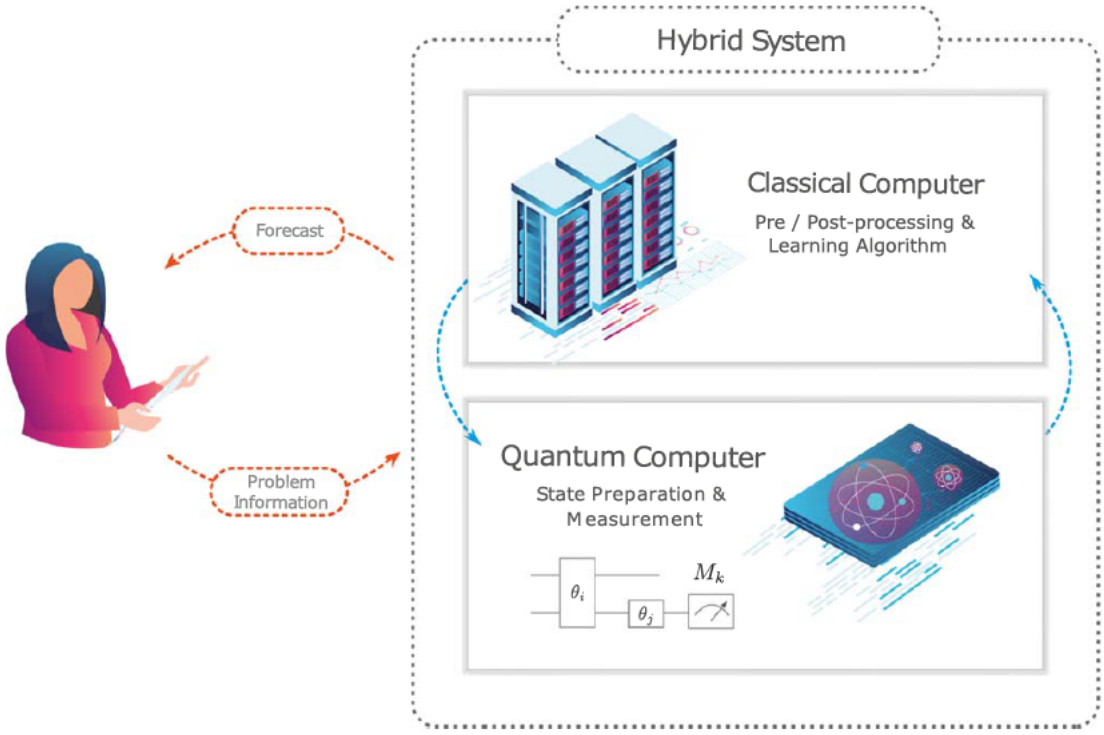

Figure 1: Depiction of hybrid algorithms. Extracted from [1].

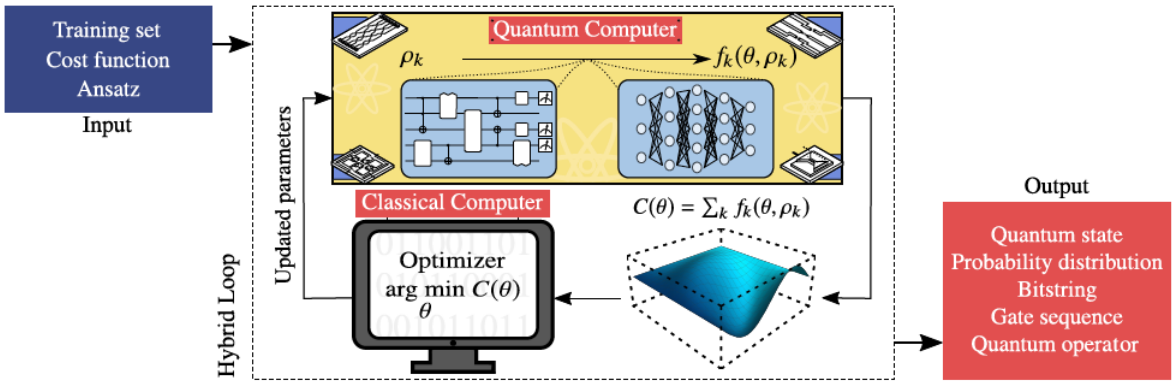

Figure 2: Schematic diagram of a Variational Quantum Algorithm (VQA). The inputs to a VQA are: (1) a set of training data $\{\rho_k\}$ used during the optimization, (2) a cost function $C(\theta)$, being θ a set of parameters that encodes the solution to the problem, and (3) an ansatz whose parameters are trained to minimize the cost. The cost is expressed in terms of some set of functions $\{f_k\}$. The ansatz is shown as a parameterized quantum circuit (left), which is analogous to a neural network (right). At each iteration of the loop, a quantum computer is used to efficiently estimate the cost. This information is fed into a classical computer that uses the power of optimizers to navigate the cost landscape $C(\theta)$ and solve the optimization problem. Once a termination condition is met, the VQA outputs an estimate of the solution (surrogate) to the problem. The form of the output depends on the concrete task that wants to be solved. The red box indicates some of the most common types of outputs. Extracted from [2].

4 Problem resolution

The PDE that will be studied in this thesis as a case study is the Heat equation. The heat equation governs heat flow and heat diffusion, as well as other diffusive processes, such as particle diffusion or the propagation of action potential in nerve cells. Hence, it is also known as the diffusion equation.

The theory of the heat equation was first developed by Joseph Fourier in 1822 for the purpose of modeling how a quantity such as heat diffuses through a given region. For heat flow, the heat equation is derived from the physical laws of conduction of heat and conservation of energy. According to Fourier's law for an isotropic medium, the rate of flow of heat energy per unit area through a surface is proportional to the negative temperature gradient across it:

$$\mathbf{q} = -\lambda \nabla u$$

where λ is the thermal conductivity of the material, $u = u(\mathbf{x}, t)$ is the temperature and $\mathbf{q} = \mathbf{q}(\mathbf{x}, t)$ is a vector field that represents the magnitude and direction of the heat flow at the point $\mathbf{x}$ of space and time t.

For the sake of simplicity, this thesis will focus on the one-dimensional heat equation. Let's study the heat flow on an insulated wire (or a thin metal rod) of uniform section and material. The wire has length L and is insulated except at the endpoints. It is initially heated to a temperature of $u_0(x)$, and the temperature distribution in the bar is $u(x, t)$. Let x denote the position along the wire and t denote time. For 1D, the equation becomes

$$\frac{q}{A} = -k \frac{\partial u}{\partial x}.$$

Let $Q = Q(x, t)$ be the internal heat energy per unit volume of the bar at each point and time. In the absence of heat energy generation, from external or internal sources, the rate of change in internal heat energy per unit volume in the material, $\frac{\partial Q}{\partial t}$ is proportional to the rate of change of its temperature, $\frac{\partial u}{\partial t}$. That is,

$$\frac{\partial Q}{\partial t} = c\rho \frac{\partial u}{\partial t}$$

where c is the specific heat capacity (at constant pressure, in case of a gas) and ρ is the density of the material [32]. In this derivation, one assumes that the material has constant density and heat capacity through space as well as time.

Applying the law of conservation of energy to a small element of the medium, and combining Fourier's law with the transferred thermal power,

$$\frac{\partial Q}{\partial t} = -\frac{\partial q}{\partial x}.$$

From the above equations it follows that

$$\frac{\partial u}{\partial t} = -\frac{1}{c\rho}\frac{\partial q}{\partial x} = -\frac{1}{c\rho}\frac{\partial}{\partial x}\left(-\lambda \frac{\partial u}{\partial x}\right) = \frac{\lambda}{c\rho}\frac{\partial^2 u}{\partial x^2}.$$

This is the heat equation in one dimension. This thesis will use a more convenient notation for partial derivatives, writing u_t instead of $\frac{\partial u}{\partial t}$ and u_{xx} instead of $\frac{\partial^2 u}{\partial x^2}$. Hence, in reduced notation the Heat equation reads:

$$u_t - w u_{xx} = 0; \tag{3}$$

where $w = \frac{\lambda}{c\rho} > 0$ is a constant (the thermal diffusivity of the material).

In order to solve the Heat equation, there are some conditions that need to be specified. First of all, fully determining the solution of an n_{th} order differential equation usually involves n arbitrary constants. These constants are determined from the boundary conditions, where the value of the solution or its derivatives is specified along the boundary of a region. When these conditions are specified as the initial value of the function and the first $n-1$ derivatives, they are called initial conditions. Finally, there are some additional conditions that the coefficient w and the data must satisfy.

- Boundary conditions (BC): the most common boundary conditions are the following.
  - Dirichlet: $u(0,t) = 0 = u(L,t)$
  - Neumann: $u_x(0,t) = 0 = u_x(L,t)$
  - Robin: $u_x(0,t) - a_0 u(0,t) = 0$ and $u_x(L,t) + a_L u(L,t) = 0$
  - Periodic: $u(-L,t) = u(L,t)$ and $u_x(-L,t) = u_x(L,t)$

  For this case, let's assume that the endpoints of the wire are kept at a fixed temperature 0 (Dirichlet BC), so that $u(0,t) = 0$ and $u(L,t) = 0$.

- Initial conditions (IC): the temperature distribution at time $t = 0$ is given by the condition $u(x,0) = u_0(x)$. For this case, let's take the function $u_0(x) = \sin(\pi \frac{x}{L})$, since it is a positive function that is zero at both $x = 0$ and $x = L$.

- Data: the value of the temperature, as well as the spatial and temporal variables, during the whole problem should be bounded in the interval $(-1, 1)$. This condition is necessary so that the data can be represented by an observable in the quantum circuit, as will be further discussed in section 4.2. Hence, the independent variables are confined in the interval $x \in (0,1)$ and $t \in (0,1)$. Regarding the temperature values, the chosen initial condition forces the temperature to reach a maximum of 1, since the sine function takes values comprised in $[-1,1]$. In fact, there is no risk that the temperature overpasses this maximum value (if surpassing 1, the data could no longer be able to be represented by the quantum circuit), due to the *maximum principle*.

  Maximum principle: if $u(x,t)$ satisfies the heat equation 3 in the closed rectangle in space-time

  $$R := \{0 < x < L, 0 < t < T\} = [0,L] \times [0,T].$$

  Then the maximum value of $u(x,t)$ over the rectangle is assumed either initially ($t = 0$) or on the lateral sides ($x = 0$ or $x = L$).

  This means that the maximum temperature in R can increase only if heat comes inside from outside R. This is a property of parabolic partial differential equations. Since there is no addition of heat from outside, the temperature of the wire will always decrease until it reaches the steady state. In this case, since the temperature u is 0 at the boundaries, it reaches its maximum value of $u_{max} = 1$ at the initial time.

- Diffusion coefficient w: it is essential that the coefficient is positive $w > 0$. A negative

parameter $w < 0$ is equivalent to reversing time and applying a back-propagation in time. Going backwards in time can lead to diverging values of the temperature and hence obtaining results $u > 1$ that the quantum circuit would be unable to represent.

All the previous conditions define an initial boundary value problem (IBVP) for the heat equation, consisting of the PDE and the domains of its independent variables, and three other conditions specified at $x = 0$, $x = L$ and $t = 0$. In the Heat equation 3, the heat diffusivity w is the unknown parameter that needs to be inferred from some data. This calibrating process has two main parts. The first part consists on, given the exact model of the PDE, building a parameterized quantum circuit that is able to solve this PDE and effectively expressing its solution. Hence, equation 3 needs to be solved. For the sake of mathematical analysis, it is sufficient to only consider the case $w = 1$. The second part makes use of the quantum circuit previously built in order to train the variational parameters: given some initial data and an optimization strategy, the circuit is able to estimate the unknown parameter w.

4.1 Solving the PDE

First of all, it is important to make a distinction between the different types of data used throughout this thesis. To this extent, the following notation will be used: u represents the real data/solution of the PDE, $\hat{u}$ represents the surrogate obtained from the solver, and u^* corresponds to the optimal solution calculated by the solver. Hence, u^* is the closest value of $\hat{u}$ that approximates the real solution u.

An arbitrary PDE can be written in the form:

$$F(u, \partial_t u, \partial_x u, \partial_x^2 u, ..., t, x) = 0 \tag{4}$$

where x and t are dependent variables and $u(x, t)$ is the sought solution.

This project considers the case of the Heat equation 3, which needs to be solved through some method in order to obtain a surrogate $\hat{u}$ which can be compared to the real data u.

Combining 4 and 3 results in the functional $F_w(u, t, x) = wu_{xx} - u_t$ corresponding to the differential equation written in the form:

$$F_w(u, t, x) = 0. \tag{5}$$

In order to solve 5, there exist 2 ways of treating w:

(A) Solver A: w is considered as a parameter; hence it is fixed. The data obtained from the solver can be expressed as $\hat{u} = solver(x, t; w)$.

(B) Solver B: w is treated as a variable; hence it is another input of the solver. The data obtained from the solver can be expressed as $\hat{u} = solver(x, t, w)$.

The optimization problem embedded in the solving process is based on the choice of a loss function that contains several terms:

- $\mathcal{L}_F$ ensures that the differential equation dynamics is fulfilled. To solve the PDE it is crucial to provide a way to quantify how well the solver trial function matches the conditions to represent the solution of the problem being considered. The classical optimizer can

then update the intrinsic parameters (θ) of the solver to reduce this distance, which corresponds to the difference between the differential equation (all terms collected on one side) and zero. This difference is estimated on a grid of N points, and is normalized by the grid size. The differential loss is defined as:

$$\mathcal{L}_F(w) = \frac{1}{N}\sum_{i=1}^{N} L(F_w(\hat{u}, t_i, x_i), 0).$$

This loss function is minimized when $F_w(\hat{u}, t_i, x_i) = 0$, i.e. when $\hat{u}(t, x, \theta)$ satisfies the partial differential equation dynamics.

- $\mathcal{L}_\Omega$ includes all the boundary conditions that have to be fulfilled in order to fully specify the PDE. It is important to check that the solution matches initial and boundary conditions. The loss is evaluated at the set of M points that form the boundary and initial points, and is normalized by the grid size. The boundary loss contribution reads:

$$\mathcal{L}_\Omega(\theta) = \frac{1}{M}\sum_{i=1}^{M} L(\hat{u}(t_i, x_i, \theta), u_i).$$

- $\mathcal{L}_d$ accounts for the fitting of the empirical dataset, evaluated at the N points of the grid.

$$\mathcal{L}_d(\theta) = \frac{1}{N}\sum_{i=1}^{N} L(\hat{u}(t_i, x_i, \theta), u_i).$$

$L(a, b)$ is a function that describes how the distance between the two arguments a and b is being measured.

The optimization process for training the model is conditioned by the solver option and depends on the treatment of w (whether as a parameter or a variable). The optimization can be done following two different procedures, according to the type of solver (A or B).

(A) Solver A: the strategy to estimate parameters is done in 2 steps, in the known two-stage method.

(i) Stage 1: Given a fixed parameter of the PDE (w), the chosen solver is applied to find the approximate solution $\hat{u}(t, x, \theta; w)$ and its derivatives parameterized by θ. The internal solver optimization works with a loss function formulated as:

$$\mathcal{L}(\theta; w) = \mathcal{L}_F(; w) + \mathcal{L}_\Omega(\theta).$$

The training over this loss function gives the optimal parameters θ^* that best reproduce the solution of the PDE. With this value of θ^*, the algorithm goes to stage (ii).

(ii) Stage 2: Once the PDE has been solved and the surrogate of the solution $\hat{u}$ has been obtained, the optimization consists on fitting the empirical dataset in order to obtain w. The PDE parameter is estimated according to the cost function:

$$\mathcal{L}(w) = \mathcal{L}_d(\theta^*(w)).$$

The training over this loss function gives the value of w that best fits the real data to the data obtained from the solver. With this value of w, the algorithm goes back to stage (i).

(B) Solver B: the optimization can be done either in 1 or 2 steps. The option of direct optimization treats the loss function as a function of both w and θ. Then, since these two parameters cannot be separated, they are optimized on an equal footing: they are trained with the same learning rate and same optimizer.

$$\mathcal{L}(\theta, w) = \mathcal{L}_F(w) + \mathcal{L}_\Omega(\theta) + \mathcal{L}_d(\theta).$$

This project will use Solver A, since it is significantly more efficient numerically to treat w as a parameter than as a variable, as will be discussed in section 4.2.3. The optimization will then be performed in two separate steps, since it gives more accurate results than optimizing all the parameters (w and θ) at the same time and it is easier to implement. Furthermore, the two-stage strategy has been widely used in parameter inferring [33].

In order to effectively train such a model, it is necessary to have access to a universal approximator $\hat{u}$ and its higher order derivatives, both with respect to inputs x and t, as well as parameters θ. As it has been seen in section 2, there are both classical and quantum methods that implement a universal approximator and allow for the resolution of the PDE. In order to establish an appropriate solver, the aim is to find an easily-differentiable surrogate to approximate the data-set that needs to be differentiated.

As studied in section 2, the mesh-based methods do not perform well due to the exponential scaling in the dimension. Therefore, it is desirable to find cost-effective alternatives such as data-driven differentiable surrogate models which can be optimized via gradient-based means. This leads to PINNs (classic alternative) or DQCs (quantum alternative). They share some similarities, since they are both universal function approximators that can be differentiated with respect to their inputs. To this extent, given these conceptual similarities between PINNs and PQCs, this project will extend the parameter inference technique to a quantum PINN analogue. To perform parameter inference on NISQ devices, the classical neural network will be replaced with a DQC. In fact, a quantum computer can naturally provide an exponential advantage in memory over classical methods.

4.2 Quantum Hardware: Differentiable Universal Function Approximator

The hybrid quantum-classical strategy starts with a quantum computer with n qubits that is assumed to be a closed quantum system, whose state can be described as a unit vector living in a complex inner product vector space $\mathbb{C}^{2n}$. The computation always starts with a state of simple preparation in the computational basis. The most used initially prepared state is $|0\rangle^{\otimes n}$ [1] (for simplicity, the tensor notation can be dropped and this state can be referred to simply as $|0^n\rangle$).

To perform quantum machine learning, first the input data has to be encoded in a quantum register. For classical data this can be achieved by using quantum feature maps, where variables x are embedded through nonlinearly transformed $\varphi(x)$ phases of rotations represented by a unitary operator $\hat{\mathcal{U}}_\varphi(x)$. Next, a linear transformation is performed using a parameterized quantum circuit $\hat{\mathcal{U}}_\theta$. Then, the unitary operator $\hat{\mathcal{U}} = \hat{\mathcal{U}}_\theta \hat{\mathcal{U}}_\varphi(x)$ is applied to the initial state, producing a new state $\hat{\mathcal{U}}\,|0^n\rangle$. Here, the value of an observable quantity can be measured. Physical observables are associated with Hermitian operators $\hat{\mathcal{C}} = \sum_i \lambda_i P_i$, where λ_i is the i^{th} eigenvalue and P_i is the projector on the corresponding eigenspace. The Born rule states that the outcome of the measurement corresponds to the one of the eigenvalues and follows the probability distribution

$p(\lambda_i) = \mathrm{Tr}(P_i\hat{\mathcal{U}} |0^n\rangle \langle 0^n| \hat{\mathcal{U}}^\dagger)$. Plugging this in the definition of expectation values yields

$$\langle \hat{\mathcal{C}} \rangle = \sum_i \lambda_i p_i(\lambda_i) = \mathrm{Tr}(\mathcal{C}\hat{\mathcal{U}} |0^n\rangle \langle 0^n| \hat{\mathcal{U}}^\dagger).$$

Next, the building blocks of the DQC will be described in detail: (1) quantum feature map, (2) variational ansatz, and (3) Hamiltonian operator for the readout of the function.

1. **Encoding**: quantum feature map

   The process of encoding classical data into a quantum state can be interpreted as a feature map from the data space to the Hilbert space of the states of n qubits. A quantum feature map is a unitary circuit $\hat{\mathcal{U}}_\varphi(x)$ that encodes the input variable x using a predefined nonlinear function φ to the amplitudes of a quantum state $\hat{\mathcal{U}}_\varphi(x) |0^n\rangle$ [1]. This is also referred to as a latent space mapping. Unlike amplitude encoding, this latent space encoding does not require access to each amplitude and is controlled by classical gate parameters, mapping the real parameter x to the corresponding variable value. Sometimes this is also called quantum embedding [3], referring to the way data is embedded in the circuit. The automatic differentiation of quantum feature maps allows derivatives to be represented as DQCs, which will be explained further in section 4.2.1.

   There are several ways to encode data into qubits and each one provides different expressive power[1]. This choice of encoding is related to kernel methods[13]. In section 2.1.2 some basis sets for encoding the surrogates were described, the most useful of them being the Chebyshev basis. Hence, the chosen map will be the Chebyshev feature map. This map belongs to the product feature map family, that uses qubit rotations and has a nonlinear dependence on the encoded variable x. In the simplest case, this corresponds to a single layer of rotations, depicted in figure 3:

   $$\hat{\mathcal{U}}_\varphi(x) = \bigotimes_{j=1}^{n} \hat{R}_{Y,j}\left(\varphi_j(x)\right)$$

[13] Methods that use kernels (or basis functions) to map the input data into a higher dimensional feature space where a specific problem may be easier to solve

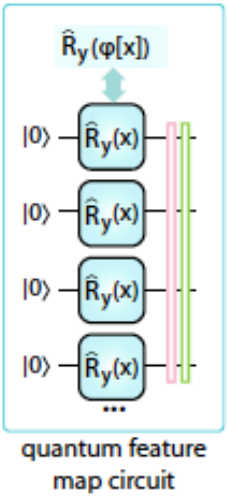

Figure 3: Chebyshev feature map, where single qubit rotations act at each qubit individually and are parametrized by a function of variable x. The thin pink block represents the variational ansatz, and the thin green block depicts the cost function measurement. The nonlinear function $\varphi(x)$ is used as an angle of rotation. Extracted from [3].

The Chebyshev feature map is characterized by the nonlinearity $\varphi(x) = 2k(j)\arccos x$, where the coefficient $k(j)$ may depend on the qubit position j. The factor 2 multiplication of $\varphi(x) = 2\arccos x$, as compared to the product map $\varphi(x) = \arccos x$, plays an important role that will be discussed in section 5.

The rotation can be expanded using Euler's formula, obtaining:

$$\hat{R}_{Y,j}(\varphi(x)) = \exp\left(-i\frac{2k\arccos x}{2}\hat{Y}_j\right) = \cos(k\arccos x)\mathbb{1}_j - i\sin(k\arccos x)\hat{Y}_j$$

. This expression can be decomposed by applying a unitary operation with matrix elements defined by degree-n Chebyshev polynomials of first and second kind [3]:

$$\hat{R}_{Y,j}(\varphi(x)) = T_k(x)\mathbb{1}_j + \sqrt{1-x^2}U_{k-1}(x)\hat{X}_j\hat{Z}_j$$

Within the Chebyshev feature maps, there are mainly two interesting types: the *sparse map* (with nonlinear function $\varphi(x) = 2\arccos x$) and the *tower map* (with nonlinear function $\varphi_j(x) = 2j\arccos x$).

Amongst all the feature maps, [3] proves that the best performing one is the Chebyshev tower feature map, since it is the most expressible and it manages to converge close to the true solution. So this will be the one implemented in this project:

$$\hat{\mathcal{U}}_\varphi(x) = \bigotimes_{j=1}^{n} \hat{R}_{Y,j}\left(2j\arccos x\right).$$

With this choice, the encoded degree grows with the number of qubits, creating a tower-like structure of polynomials with increasing $k = j$. Furthermore, this choice allows to obtain large expressibility without increasing the system size and number of rotations.

The nonlinear function $\arccos(x)$ is defined in the domain $\Omega \in (-1, 1)$. This implies that the independent variables x and t of equation 3 that need to be encoded must be confined within the interval $(-1, 1)$.

The state $\hat{\mathcal{U}}_\varphi(x)\,|0^n\rangle$ represents a basis of n Chebyshev polynomials which can be read out by measuring for example in the Z basis on each qubit. With a subsequent variational circuit (with parameters θ) the Chebyshev basis functions are combined. In the limit of large and controlled entanglement, such a setup allows access to up to $\mathcal{O}(2^n)$ Chebyshev polynomials, due to the chaining and nesting properties of products of these polynomials.

2. **Variational Quantum Algorithm**

The variational quantum circuit (typically referred to as a variational quantum ansatz) allows to manipulate the latent space basis function and bring the derivatives and function to the required form. The ansatz is parameterized by a vector of variational parameters θ that can be adjusted in a quantum-classical optimization loop. These parameters are encoded in the unitary $\hat{\mathcal{U}}_\theta(\theta)$.

The most common ansatz structure used in machine learning problems is the Hardware Efficient Ansatz (HEA) [1] with layers of parameterized rotations followed by layers of CNOT gates between neighboring qubits. The structure of a HEA quantum circuit corresponds to concatenated layers of single qubit rotations and global entangling layers for all n qubits, shown schematically in figure 4. Rotations are arranged in a $\hat{R}_Z - \hat{R}_X - \hat{R}_Z$ sequence parameterized by independent angles θ such that arbitrary single-qubit operations can be reproduced. The entangling layer is chosen as a network of CNOTs. The block of rotations plus CNOTs is then repeated for a depth of d times. As the number of layers d grows, the circuit's expressive power[14] increases. However, this comes with an increased number of controlled parameters which can complicate the search for an optimal θ^* for the solution, which can be a problem in trainability.

[14]The expressive power is the ability to represent arbitrary n-qubit unitary gates

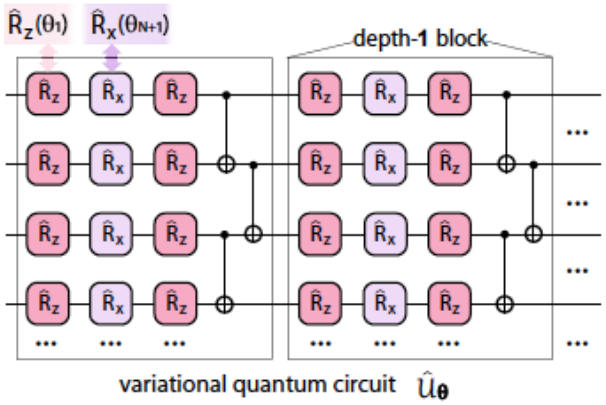

Figure 4: Hardware Efficient Ansatz. It consists of a parameterized rotation layer forming $\hat{R}_Z - \hat{R}_X - \hat{R}_Z$ pattern, such that an arbitrary single qubit rotation can be implemented. Variational angles θ are set for each rotation individually. The rotation layer is then followed by an entangling layer chosen as CNOT operations between nearest neighbours. The blocks of "rotations-plus-entangler" are repeated d times to form the full variational circuit $\hat{\mathcal{U}}_\theta$. Extracted from [3].

Variational quantum algorithms that are used for quantum machine learning rely on the ability to automatically differentiate parameterized quantum circuits with respect to underlying parameters, thanks to the parameter shift rule, which will be discussed in section 4.2.1.

3. **Output**

   The output of the circuit $\hat{u}(t, x, \theta)$ is read out as an expectation value of a predefined Hermitian cost operator $\hat{C}$, which is measured as an observable. In general there are many possible choices of cost operators. The simplest example corresponds to the magnetization of a single qubit j, $\langle \hat{Z}_j \rangle$. This choice allows to represent functions in the range $[-1, 1]$ and requires rescaling for other intervals. This implies equation 3 to be confined within the interval $(-1, 1)$. Then, the chosen operator corresponds to the total magnetization in the Z direction $\hat{C} = \sum_{j=1}^{n} \hat{Z}_j$. Since the unitary observable is the Pauli matrix $\hat{Z}$, the measured values will be bounded between -1 and 1. In this way, the data readout problem is avoided, since the solution is encoded in the observable of an operator, such that the expectation can be routinely calculated.

The final universal approximator for the scalar input variables x and t and variational parameters θ reads:

$$\hat{u}(t, x, \theta) = \langle 0^n | \hat{\mathcal{U}}_\varphi^\dagger(t, x) \hat{\mathcal{U}}_\theta^\dagger(\theta) \hat{C} \hat{\mathcal{U}}_\theta(\theta) \hat{\mathcal{U}}_\varphi(t, x) | 0^n \rangle . \tag{6}$$

4.2.1 Parameter shift rule

For solving PDEs it is necessary to access the derivatives of the circuits representing the functions $\hat{u}_t \equiv \frac{\partial \hat{u}}{\partial t}$, $\hat{u}_{tt} \equiv \frac{\partial^2 \hat{u}}{\partial t^2}$, $\hat{u}_x \equiv \frac{\partial \hat{u}}{\partial x}$ and $\hat{u}_{xx} \equiv \frac{\partial \hat{u}}{\partial x}$. This can be done using automatic differentiation techniques for quantum circuits through the parameter shift rule.

The output of the variational circuit $\hat{u}(t, x, \theta)$ 6 (i.e., the expectation of the observable) can be

written as a "quantum function"

$$\hat{u}(t,x) = \langle 0^n | \hat{\mathcal{U}}_{\varphi}^{\dagger}(t,x) \hat{M} \hat{\mathcal{U}}_{\varphi}(t,x) | 0^n \rangle \; ; \; \hat{M} = \hat{\mathcal{U}}_{\theta}^{\dagger}(\theta) \hat{\mathcal{C}} \hat{\mathcal{U}}_{\theta}(\theta).$$

The partial derivatives of $\hat{u}(t,x)$ can be expressed as a linear combination of other quantum functions that differ only in a shift of the argument. This means that partial derivatives of a variational circuit can be computed by using the same variational circuit architecture.

The crucial step of the algorithm is the differentiation of the quantum feature map circuit, $\frac{d\hat{\mathcal{U}}_{\varphi_t}(t)}{dt}$ and $\frac{d\hat{\mathcal{U}}_{\varphi_x}(x)}{dx}$, which is the part that depends on the variables x and t. Analytic derivatives of quantum circuits can be estimated by measuring overlaps between quantum states, and are proven to benefit circuit optimization. This strategy is known as parameter shift rule [34] [35] [36] [37] [38].

As an example, the derivatives will be computed using the parameter shift rule for the input variable x. As a first step, the overall unitary $\hat{\mathcal{U}}_{\varphi}^{\dagger}(t,x)$ that encodes the variable x is generated by a sequence of single-parameter gates of Y-rotations $\hat{R}_{Y,j}(\varphi_j(x))$. Hence, equation 6 is fully analytically differentiable via the parameter shift rule, which requires two evaluations of the same circuit with shifted angles for each circuit-level parameter. For 1-qubit Pauli rotation gates, the parameter shift rule is derived in Annex 9.2.

Let's define the quantum function

$$f(\varphi_j) = \langle 0^n | \bigotimes_{j=1}^{n} \hat{R}_{Y,j}^{\dagger}(\varphi_j) \hat{M} \bigotimes_{j=1}^{n} \hat{R}_{Y,j}(\varphi_j) | 0^n \rangle \; ; \; \hat{M} = \hat{\mathcal{U}}_{\theta}^{\dagger}(\theta) \hat{\mathcal{C}} \hat{\mathcal{U}}_{\theta}(\theta).$$

Index j runs through the individual quantum operations used in the feature map encoding.

The first derivative reads:

$$\frac{\partial \hat{u}}{\partial x} = \frac{1}{2} \sum_{j_1=1}^{n} \varphi'_{j_1}(x) \left(F^{+}_{j,j_1} - F^{-}_{j,j_1} \right) \tag{7}$$

with $F^{+}_{j,j_1} = f(\varphi_j + \frac{\pi}{2}\delta_{jj_1})$ and $F^{-}_{j,j_1} = f(\varphi_j - \frac{\pi}{2}\delta_{jj_1})$.

Figure 5 shows the shifted angles for each qubit.

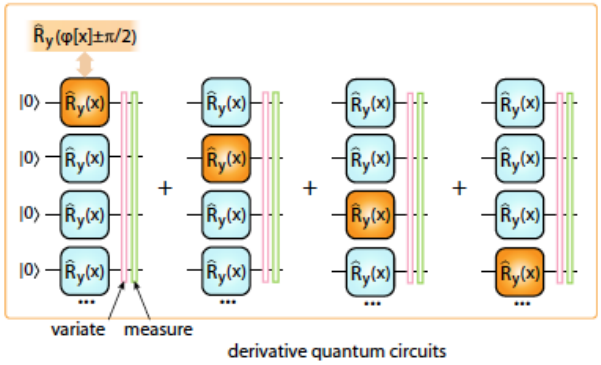

Figure 5: Derivative quantum circuit for the Chebyshev feature map. Differentiation over variable x follows the chain rule, with the expectation value of the derivative written as a sum of separate expectations with shifted phases, repeated for each x-dependent rotation. Extracted from [3].

Applying the parameter shift rule once again allows to obtain the second-order derivative with four shifted terms for each generator.

$$\frac{\partial^2 \hat{u}}{\partial x^2} = \frac{1}{2}\sum_{j_1=1}^{n} \varphi''_{j_1}(x)\left(F^{+}_{j,j_1} - F^{-}_{j,j_1}\right) + \frac{1}{4}\sum_{j_1=1}^{n}\sum_{j_2=1}^{n} \varphi'_{j_1}(x)\varphi'_{j_2}(x)\left(F^{++}_{j,j_1,j_2} - F^{+-}_{j,j_1,j_2} - F^{-+}_{j,j_1,j_2} + F^{--}_{j,j_1,j_2}\right) \tag{8}$$

with

$$F^{++}_{j,j_1,j_2} = f(\varphi_j + \frac{\pi}{2}(\delta_{jj_1} + \delta_{jj_2}))$$
$$F^{+-}_{j,j_1,j_2} = f(\varphi_j + \frac{\pi}{2}(\delta_{jj_1} - \delta_{jj_2}))$$
$$F^{-+}_{j,j_1,j_2} = f(\varphi_j + \frac{\pi}{2}(-\delta_{jj_1} + \delta_{jj_2}))$$
$$F^{--}_{j,j_1,j_2} = f(\varphi_j + \frac{\pi}{2}(-\delta_{jj_1} - \delta_{jj_2})).$$

If implemented naively, the second derivative requires $2n+4n^2$ evaluations of circuit expectation values. This can be reduced by making use of symmetries in the shifted expectation values and reusing previously calculated values, i.e. those for the function and its first-order derivative. The reduced number of additional circuit evaluations required for the calculation of the second derivative is $2n^2$. For instance, the total number of possible shifts is limited to four, due to the periodicity of the rotation angles. Hence one can define a shift basis as $s = \{0, \frac{\pi}{2}, \pi, \frac{-\pi}{2}\}$.

When increasing the order of the derivative, several combinations of delta functions appear in the analytical expression 8. Let's denote Δ the sum of deltas for a given derivative order (e.g. $\Delta = \delta_{jj_1} - \delta_{jj_2} + \delta_{jj_3} - \delta_{jj_4}$ for one term of the 4^{th}-derivative). Then, all the shifts are multiples of $\Delta\frac{\pi}{2}$, $\Delta = 0, 1, 2, ...$ Table 2 resumes the corresponding shift for every value of Δ.

The derivatives of the non-linear function $\varphi_j(x) = 2j \arccos x$ are computed as follows:

$$\varphi'_j(x) = 2j\frac{-1}{\sqrt{1-x^2}};\ \varphi''_j(x) = 2j\frac{-x}{(1-x^2)^{3/2}}$$

$\Delta \mod 4$	**Shift**
0	0
1	$\pi/2$
2	π
3	$-\pi/2$

Table 2: Shift for every combination of deltas

The DQC technique can straightforwardly be extended to multiple variables $(x, y, t...)$ by using multiple feature maps, one for each variable, and differentiating only those feature maps relevant to the variable differentiation of interest.

The parameter shift rule is designed to estimate an analytic (exact, zero-bias) gradient via the measurement of expectation values, with quantum gate parameters being shifted to different values (originally considered to be fixed). This algorithm uses automatic differentiation (AD) to calculate all the spatial and temporal derivatives, returning machine-precision derivatives. This approach is considerable more accurate than any form of numerical differentiation. Since automatic differentiation provides an analytical derivative of the circuit at any value of variables t and x, this scheme does not include the accumulated error from approximating the derivatives that occurs when using other typical solvers that involve numerical differentiation, such as Euler's method and finite differences, that suffer from imprecision errors.

This is the standard parameter shift rule, which is valid only for the specific type of generators that are involutory (i.e. $\hat{G}^2 = \mathbb{1}$) and idempotent operators ($\hat{G}^2 = \hat{G}$, for example projectors). The rules work for unitaries generated by operators with no more than two unique eigenvalues in the spectrum, as derived in section 9.2. Since the generator used in this work is based on single-qubit rotations, the standard parameter shift rule is completely valid. In order to increase the expressivity of the quantum universal approximator, it could be highly beneficial to use a more intricate feature map. However, for arbitrary generators (and hence for arbitrary and more complicated feature maps) with a generic non-degenerate spectrum, the parameter shift rule no longer holds, and the rules have to be generalized as in [35]. This derivative evaluation can be performed for instance using a spectral decomposition, where the derivative corresponds to the weighted sum of measured expectations for circuits with shifted parameters. Then, the number of function evaluations is equal to the number of unique positive nonzero spectral gaps (eigenvalue differences) for the generator.

4.2.2 Different approaches for implementing the circuit

This section explores a more technical approach for implementing the circuit that is alternative to the general one. The general approach treats the gates as matrices and performs tensor products and matrix multiplication in order to obtain the general matrix that describes the full circuit. If working with n qubits, the matrix of the system has a size of $2^n \times 2^n$. The major caveat of this technique comes when increasing the number of qubits, because the size of the matrices will grow exponentially according to the number of qubits used. Then, performing matrix multiplication is a high cost operation. In fact, during the experimental implementation of the code, the number of qubits used was found to be a bottleneck because the computational time increased exponentially. In order to solve this, another approach was considered based on matrix decomposition.

In section 4.2.1 it was seen that the output of the circuit could be expressed as

$$f(\varphi_j) = \langle 0^n | \bigotimes_{j=1}^{n} \hat{R}^{\dagger}_{Y,j}(\varphi_j) \hat{M} \bigotimes_{j=1}^{n} \hat{R}_{Y,j}(\varphi_j) \, |0^n\rangle \; ; \; \hat{M} = \hat{\mathcal{U}}^{\dagger}_{\theta}(\theta) \hat{\mathcal{C}} \hat{\mathcal{U}}_{\theta}(\theta).$$

Matrix $\hat{M}$ has a size of $2^n \times 2^n$, but it can be expressed by its decomposition into the Pauli basis. That is,

$$\hat{M}(\theta) = \sum_{i=0}^{4^n-1} c_i \sigma_i$$

where c_i are real coefficients and σ_i are Pauli operators, $i_j \in \{\emptyset, X, Y, Z\}$.

A given coefficient c_k can be found through

$$\mathrm{Tr}\left[\sigma_k \cdot \hat{M}\right] = \sum_i c_i \mathrm{Tr}\left[\sigma_k \cdot \sigma_i\right].$$

Since Pauli matrices are orthogonal

$$\sigma_k \cdot \sigma_i = \delta_{k,i} \mathbb{1},$$

then

$$\mathrm{Tr}\left[\sigma_k \cdot \hat{M}\right] = \sum_i c_i \delta_{k,i} \mathrm{Tr}\left[\mathbb{1}\right].$$

Since $\mathrm{Tr}\left[\mathbb{1}\right] = 2^n$, one gets the final result:

$$c_k = \frac{1}{2^n} Tr\left[\sigma_k \cdot \hat{M}\right].$$

Computing the trace of $f(\varphi_j)$ one gets:

$$f(\varphi_j) = \mathrm{Tr}[f(\varphi_j)] = \sum_{i=0}^{4^n-1} c_i \mathrm{Tr}\left[|0^n\rangle \langle 0^n| \bigotimes_{j=1}^{n} \hat{R}^{\dagger}_{Y,j}(\varphi_j) \bigotimes_{j=1}^{n} \sigma_{i_j} \bigotimes_{j=1}^{n} \hat{R}_{Y,j}(\varphi_j) \right].$$

Applying the property $\mathrm{Tr}[\hat{A} \otimes \hat{B}] = \mathrm{Tr}[\hat{A}] \cdot \mathrm{Tr}[\hat{B}]$, one gets

$$f(\varphi_j) = \sum_{i=0}^{4^n-1} c_i \prod_{j=1}^{n} \langle 0| \hat{R}^{\dagger}_{Y,j}(\varphi_j) \sigma_{i_j} \hat{R}_{Y,j}(\varphi_j) \, |0\rangle .$$

The main advantage of using this approach is that instead of working with a Pauli string σ_i of size $2^n \times 2^n$, one works with a single-qubit Pauli matrix σ_{i_j} of size 2×2.

With this approach, the tensor product of matrices gets reduced to a multiplication of 3 single-qubit matrices: $\hat{R}^{\dagger}_{Y} \sigma_{i_j} \hat{R}_{Y}$. The tensor product produces matrices of size $2^n \times 2^n$ with a high level of sparsity, so most of the entries are 0 values and contain no information. To this extent, it is far more advantageously to work with 2×2 matrices due to the less number of entries, which

reduces the computational cost of storing data in the matrix elements.

This approach is also very useful when dealing with the shifts introduced by the differentiation of the outputs. As discussed in section 4.2.1, the calculation of the first and second derivative involves the circuit evaluation $f(\varphi + \Delta\frac{\pi}{2})$, $\Delta = 0, 1, 2....$ Due to the periodicity of the rotation angles, this expression can be simplified to $f(\varphi + s)$, $s = \{0, \frac{\pi}{2}, \pi, \frac{-\pi}{2}\}$. Hence it is sufficient to analyze just 4 different cases, as the following change of variable applies: $s = (\Delta \mod 4)\frac{\pi}{2}$.

Remembering the expression of Y-Pauli rotation gate:

$$\hat{R}_Y(\varphi) = e^{-\frac{i}{2}\varphi Y}.$$

When applying a shift one gets:

$$\hat{R}_Y(\varphi + s) = e^{-\frac{i}{2}(\varphi+s)Y} = e^{-\frac{i}{2}\varphi Y} \cdot e^{-\frac{i}{2}sY} = \hat{R}_Y(\varphi)\hat{R}_Y(s).$$

Then, for any shift s, the matrix multiplication that needs to be performed is the following:

$$\hat{R}_Y^\dagger(\varphi + s)\sigma_{i_j}\hat{R}_Y(\varphi + s) = \hat{R}_Y^\dagger(\varphi)\hat{R}_Y^\dagger(s)\sigma_{i_j}\hat{R}_Y(s)\hat{R}_Y(\varphi).$$

Since the shift s is a multiple of $\frac{\pi}{2}$, the following transformation holds:

$$\hat{T} = \hat{R}_Y^\dagger(s)\sigma_{i_j}\hat{R}_Y(s) \longrightarrow \sigma_{i_k}.$$

That means that every Pauli matrix maps into another Pauli matrix, depending on the value of the shift. The corresponding transformations are given in table 3.

s	$\sigma_{i_j} = \mathbb{1}$	$\sigma_{i_j} = X$	$\sigma_{i_j} = Y$	$\sigma_{i_j} = Z$
0	$\mathbb{1}$	X	Y	Z
$\pi/2$	$\mathbb{1}$	Z	Y	-X
π	$\mathbb{1}$	-X	Y	-Z
$-\pi/2$	$\mathbb{1}$	-Z	Y	X

Table 3: Values of $\hat{T}$ according to the transformation given by the shift and the Pauli matrix

According to table 3, applying a shift translates into mapping the shift and initial Pauli matrix into another Pauli matrix.

$$\hat{R}_Y^\dagger(\varphi + s)\sigma_{i_j}\hat{R}_Y(\varphi + s) = \hat{R}_Y^\dagger(\varphi)\sigma_{i_k}\hat{R}_Y(\varphi).$$

This transformation simplifies the calculations. However, the drawback of this approach is that this decomposition contains many terms. In this case, the number of distinct Pauli strings scales as 4^n where n is the number of qubits.

4.2.3 Encoding for multiple variables

To solve the Heat equation, the UFA has two inputs (t, x) and one output $\hat{u}$. Therefore, every input variable needs a feature map for its encoding. In this case, every variable is encoded with the same feature map, corresponding to a R_Y rotation.

The global feature map can be built either stacking the previous maps onto one circuit in parallel or in series. In the parallel case, the global feature map reads:

$$\hat{\mathcal{U}}_\varphi(t,x) = \hat{\mathcal{U}}_{\varphi_t}(t) \bigotimes \hat{\mathcal{U}}_{\varphi_x}(x) = \Big(\bigotimes_{j=1}^{n/2} \hat{R}_{Y,j}\left(\varphi_j(t)\right) \Big) \bigotimes \Big(\bigotimes_{j=1}^{n/2} \hat{R}_{Y,j}\left(\varphi_j(x)\right) \Big).$$

In the series case, the global feature map reads:

$$\hat{\mathcal{U}}_\varphi(t,x) = \hat{\mathcal{U}}_{\varphi_t}(t) \cdot \hat{\mathcal{U}}_{\varphi_x}(x) = \Big(\bigotimes_{j=1}^{n} \hat{R}_{Y,j}\left(\varphi_j(t)\right) \Big) \cdot \Big(\bigotimes_{j=1}^{n} \hat{R}_{Y,j}\left(\varphi_j(x)\right) \Big).$$

In this project, the chosen feature stacking is the parallel one [39], since it treats the independent variables (t, x) separate from one another. The main advantage of using the parallel map is that every variable map acts on its own qubits, which do not interact with the qubits of the other variable map; thus separating independent variables.

On the other hand, the series stacking can also be an option [40]. In this case, acting on the initial state, every feature map uses a different rotation: one for the time-dependence embedding and the other for the latent variable embedding.

When working with multiple variables, the main limitation comes when increasing the number of dimensions, a phenomenon known as the *curse of dimensionality*. Every independent variable added in the circuit needs its own encoding. In a parallel setting, this implies increasing the number of qubits, gates and evaluations of the circuit, since the derivatives of the input variable need to be calculated. This translates into an increase in the computational cost and time. As mentioned in section 4.1, this is the main reason for treating w as a parameter rather than as a variable, since it would be an extra input in the DQC that would need its own encoding.

4.3 Classical hardware: optimization process for inferring the PDE parameters

In the example of this work, the Heat equation with scalar variable u is parameterized with the parameter w to be identified. The coefficient w must be inferred from some given data-set. In the Heat equation it is important not to introduce a trainable coefficient to u_t, since it would allow equation 5 to be satisfied trivially with coefficients as zero.

The classical part of the hybrid quantum-classical algorithm consists on performing an optimization in 2 stages, as resolved in section 4.1: (1) obtaining the optimal parameters θ^* that allow to express the solution and (2) use this optimal circuit to train a particular loss function and estimate the unknown parameter w^*.

4.3.1 Step 1: solving the PDE

Just like classical models, PQC models are trained to perform data-driven tasks. The task of learning an arbitrary function from data is mathematically expressed as the minimization of a

loss function $\mathcal{L}(\theta)$, also known as the objective function, with respect to the parameter vector θ. According to section 4.1, the parameters θ (phases of unitary operators that form the circuit) are adjusted variationally based on a training objective defined by the loss function:

$$\mathcal{L}_\theta(\theta) = \lambda_f \mathcal{L}_f(\theta) + \lambda_b \mathcal{L}_b(\theta); \quad \lambda_f, \lambda_b > 0.$$

As discussed in section 4.1, w is passed as a fixed parameter and is used to calculate the first term of the loss function, which makes use of w in order to fit the PDE (with N being the total number of points in the data-set):

$$\mathcal{L}_f(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(w\hat{u}_{xx}(t_i, x_i, \theta) - \hat{u}_t(t_i, x_i, \theta), 0).$$

The second term of the loss function accounts for boundary conditions (with M being the number of boundary and initial points):

$$\mathcal{L}_b(\theta) = \frac{1}{M} \sum_{i=1}^{M} L(\hat{u}(t_i, x_i, \theta), u_i).$$

Each term of the loss function has some coefficient $\lambda > 0$ that controls the weight of its contribution in the optimization procedure. In particular, the boundary terms play an important role, since larger boundary weights ensure that the boundary is prioritised and represented to higher precision. In order to have more control over the contribution of the boundaries, their loss function will be split and each boundary will be treated separately. This allows to adjust each contribution independently of the others in a decoupled system.

$$\mathcal{L}_b(\theta) = \lambda_{bl} \mathcal{L}_{X_l}(\theta) + \lambda_{br} \mathcal{L}_{X_r}(\theta) + \lambda_{b0} \mathcal{L}_{T_0}(\theta).$$

- Left boundary: $\mathcal{L}_{X_l}$

  $$\mathcal{L}_{X_l}(\theta) = \frac{1}{M_{X_l}} \sum_{i=1}^{M_{X_l}} L(\hat{u}(t_i, 0, \theta), u_i)$$

  with M_{X_l} being the points with $x = 0$.

- Right boundary: $\mathcal{L}_{X_r}$

  $$\mathcal{L}_{X_r}(\theta) = \frac{1}{M_{X_r}} \sum_{i=1}^{M_{X_r}} L(\hat{u}(t_i, L, \theta), u_i)$$

  with M_{X_r} being the points with $x = L$.

- Initial boundary: $\mathcal{L}_{T_0}$

  $$\mathcal{L}_{T_0}(\theta) = \frac{1}{M_{T_0}} \sum_{i=1}^{M_{T_0}} L(\hat{u}(0, x_i, \theta), u_i)$$

  with M_{T_0} being the points with $t = 0$.

This thesis considers two approaches for calculating the weights of each loss term: λ_f, λ_{bl}, λ_{br} and λ_{b0}. In both approaches the weights are normalized so that $\sum_i \lambda_i = 1$.

1. Approach 1: it gives more importance to the boundary terms.

   Each term is calculated as $\lambda_i = \frac{1/N_i}{\sum_i 1/N_i}$ being N the number of points.

2. Approach 2: it gives more importance to the interior points of the data-set.

   Each term is calculated as $\lambda_i = \frac{N_i}{\sum_i N_i}$ being N the number of points.

The exact expressions for each approach are gathered in Table 4. Depending on the occasion, it might be interesting to reinforce the boundary and initial conditions (hence using approach 1 - Λ_1) or the interior points of the domain (hence using approach 2 - Λ_2).

Weights	Λ_1	Λ_2
λ_f	$\frac{1/N}{1/N+1/M_{X_l}+1/M_{X_r}+1/M_{T_0}}$	$\frac{N}{N+M_{X_l}+M_{X_r}+M_{T_0}}$
λ_{bl}	$\frac{1/M_{X_l}}{1/N+1/M_{X_l}+1/M_{X_r}+1/M_{T_0}}$	$\frac{M_{X_l}}{N+M_{X_l}+M_{X_r}+M_{T_0}}$
λ_{br}	$\frac{1/M_{X_r}}{1/N+1/M_{X_l}+1/M_{X_r}+1/M_{T_0}}$	$\frac{M_{X_r}}{N+M_{X_l}+M_{X_r}+M_{T_0}}$
λ_{b0}	$\frac{1/M_{T_0}}{1/N+1/M_{X_l}+1/M_{X_r}+1/M_{T_0}}$	$\frac{M_{T_0}}{N+M_{X_l}+M_{X_r}+M_{T_0}}$

Table 4: Different approaches for calculating the weights of each loss term

Then, the loss function that will be used for the optimization reads:

$$\mathcal{L}_\theta(\theta) = \lambda_f \mathcal{L}_f(\theta) + \lambda_{bl}\mathcal{L}_{X_l}(\theta) + \lambda_{br}\mathcal{L}_{X_r}(\theta) + \lambda_{b0}\mathcal{L}_{T_0}(\theta).$$

The best choice for the parameters that allows the DQC to express the solution of the Heat equation corresponds to:

$$\theta^* = \arg\min_\theta \mathcal{L}_\theta(\theta).$$

However, this is a highly non-convex optimization. This is the main reason why the classical optimization problems associated with VQAs are expected to be NP-hard in general, because they involve cost functions that can have many local minima and the optimizer can be trapped in those local minima, thus not providing with the actual solution.

This thesis considers several choices of the loss defined by three distance definitions L.

(A) Mean Square Error (MSE): $L(a,b) = (a-b)^2$

(B) Mean Absolute Error (MAE): $L(a,b) = |a-b|$

(C) Maximum Absolute Error (MAX): the definition of the loss function is different, since it computes a maximum rather than a mean. Given $\mathbf{a} = (a_1, ...a_N)$ and $\mathbf{b} = (b_1, ...b_N)$, the loss function defined by the MAX approach is given by:

$$\mathcal{L} = (\max\{|a_i - b_i|: \ \ 1 \leq i \leq N\})^2.$$

The choice of loss functions dictates how the optimizer perceives the distance between vectors and therefore affects the convergence. MSE places a greater emphasis on larger distances and

smaller weight on small distances, strongly discouraging terms with large L. Both MAE and MAX do not place such an emphasis and may have slower convergence. However, once close to the optimal solution they can achieve higher accuracy than MSE.

There are two deterministic approaches to optimization problems:

- First-order derivative methods (gradient descent method): they rely on following the derivative (or gradient) downhill/uphill to find the optimal solution.
- Second-order derivative methods (Newton's method): they are based on the Hessian, the matrix containing the second derivatives. They require the calculation of the inverse of the Hessian, which can be a computationally expensive operation.

The optimizer chosen for obtaining the variational parameters θ^* is the L-BFGS-B, since it is a widely-used algorithm for parameter estimation in machine learning. This algorithm has several variants:

- Broyden–Fletcher–Goldfarb–Shanno (BFGS): it is a second-order optimization algorithm that belongs to the class of quasi-Newton methods. The quasi-Newton methods approximate the inverse of the Hessian (instead of intensively calculating it as Newton methods) using the gradient, which is calculated by finite-differences [41]. Hence the BFGS method updates the calculation of the Hessian matrix at each iteration rather than precisely recalculating it for each iteration of the algorithm.
- Limited-memory BFGS (L-BFGS): it approximates the BFGS algorithm using a limited amount of computer memory. In the BFGS algorithm, the size of the Hessian and its inverse depends on the number of input parameters to the objective function. When dealing with a very large number of variables, the size of the Hessian becomes extremely large. The L-BFGS alleviates this problem by assuming a simplification of the inverse of the Hessian in the previous iteration. Since the DQC can have a lot of parameters θ to be optimized (depending on the number of qubits and the depth of the circuit), the L-BFGS is a better choice than BGFS.
- L-BFGS-B: it extends L-BFGS to handle simple box constraints (bound constraints) on variables; that is, lower and upper bounds. The method works by identifying fixed and free variables at every step (using a simple gradient method), and then using the L-BFGS method on the free variables only to get higher accuracy, and then repeating the process. Since the θ parameters of the circuit are rotation angles, they are bounded in the interval $\theta \in [0, 2\pi]$.

The L-BFGS-B optimizer computes the gradient of the loss function with respect to the variational parameters θ and updates these angles from iteration $n_j = 1$ into the next one n_{j+1},

$$\theta^{n_{j+1}} \leftarrow \theta^{n_j} - \alpha \nabla_\theta \mathcal{L}_\theta$$

with α being a learning rate (a hyperparameter controlling the magnitude of the update). $\nabla_\theta \mathcal{L}_\theta$ is the gradient vector, whose partial derivatives are calculated numerically using a finite difference scheme. The optimizer repeats this steps until it reaches the exit condition. The exit condition and the convergence criterion of the optimizer depend on its intrinsic characteristics [42]. For this problem, they have been settled as follows:

- ftol= $1e^{-9}$ $\longrightarrow$ tolerance for the value of the loss function.
- gtol= $1e^{-5}$ $\longrightarrow$ tolerance for the value of the loss gradient.
- eps= $1e^{-8}$ $\longrightarrow$ the absolute step size used for numerical approximation of the gradient using forward differences.
- maxfun= 15000 $\longrightarrow$ maximum number of function evaluations
- maxiter= 15000 $\longrightarrow$ maximum number of iterations reached

The optimizer trains over the variational parameters θ, which are set to initial values θ_0 as random angles contained in the interval $[0, 2\pi]$. With a random initial guess, the circuit obtains a good expressivity. Alternatively, in the case where the DQC was not able to correctly express the solution of the PDE, the technique of regularization can be used to achieve a good initial guess. The regularization procedure [3] helps the optimizer to avoid getting trapped in local minima. However, the built DQC was able to provide good optimal values θ^*, so the regularization technique was not necessary.

There are other parameters that play an important role in the expressivity of the DQC. For instance, the number of qubits (n) and the depth or number of layers of the variational circuit (d). The initial thought is that increasing the size of the DQC will result in an increase of the expressivity of the circuit. The influence of this parameters will be discussed in section 5.

4.3.2 Step 2: inferring the correct parameters

For a given w, step 1 obtains the optimal parameters $\theta^*(w)$. According to the solving strategy discussed in section 4.1, the second stage is to fit the empirical data-set with the surrogate $\hat{u}(\theta^*)$ obtained from the circuit in the previous stage. From this fit, the new parameter w can be inferred.

The fitting is adjusted through the loss function

$$\mathcal{L}_w(w) = \mathcal{L}_d(\theta^*(w)) = \frac{1}{N}\sum_{i=1}^{N} MSE(\hat{u}(t_i, x_i, \theta^*(w)), u_i).$$

The optimal value of w for the given θ^* corresponds to the value that minimizes this loss function.

$$w^* = \arg\min_w \mathcal{L}_w(w).$$

In stage 1 of the optimization process, the variable passed to the optimizer (θ) is a vector with several components. However, in this second stage the optimization is performed over a single variable w which is one-dimensional. Hence, one does not need to use such complex optimizer as L-BFGS-B, that allows for the optimization over a vector of variables. Instead, it is enough to use a 1D scalar optimizer. Some of the most used optimizers for scalar functions are the following:

- Golden: this method uses the golden section search technique. The Golden-Search method minimizes a one-dimensional function $f(x)$ on the initially defined interval $[a, b]$. This

method uses the analog of the bisection search to decrease the bracketed interval. Golden-Search however divides the interval $[a_i, b_i]$ into two sub-intervals by using the points $x_{i,1}$ and $x_{i,2}$ with

$$x_{i,j} = a_i + \alpha_j(b_i - a_i); \;\; \alpha_1 = \frac{3-\sqrt{5}}{2}; \;\; \alpha_2 = 1 - \alpha_1 = \frac{\sqrt{5}-1}{2}.$$

The values α_1 and α_2 are chosen in a way such that the interval $[a_i, b_i]$ is intersected according to the golden ratio. The function values $f(x_{i,j})$ are computed for $j = 1, 2$ and are compared so that the next iteration's interval is chosen according to this comparison:

- If $f(x_{i,1}) < f(x_{i,2}) \longrightarrow a_{i+1} = a_i; \; b_{i+1} = x_{i,2}$
- If $f(x_{i,1}) > f(x_{i,2}) \longrightarrow a_{i+1} = x_{i,1}; \; b_{i+1} = b_i$

The problem with Golden-search is its slow convergence.

- Brent: this method uses Brent's algorithm to find a local minimum of a scalar function $f(x)$. The algorithm uses inverse parabolic interpolation when possible to speed up the convergence of the golden section method. It relies on parabolic approximations of the function $f(x)$. If a parabola is given by $Ax^2 + Bx + C$, then its minimum is located at $x = -\frac{B}{2A}$. The method starts with the initial interval $[a, b]$ and computes the intersection point $x_1 = \frac{a+b}{2}$. The method then computes a parabola that contains exactly the three points $(a, f(a)), (b, f(b)), (x_1, f(x_1))$. The minimum of this parabola is calculated as:

$$x_2 = b - \frac{1}{2}\frac{(b-a)^2\{f(b) - f(x_1)\} - (b-x_1)^2\{f(b) - f(a)\}}{(b-a)\{f(b) - f(x_1)\} - (b-x_1)\{f(b) - f(a)\}}.$$

  Then b is replaced with x_2 and a new parabola is computed through the new points. The method is repeated until convergence is reached. In every iteration step, only one query to the function is needed (the value at the new point), in contrast to the two queries needed in the Golden method.

- Bounded: it can perform bounded minimization. It uses the Brent method to find a local minimum in the interval $x_1 < x_{opt} < x_2$.

Since the w variable is not bounded (only lower bounded, since $w > 0$), the chosen method to minimize the scalar function $\mathcal{L}(w)$ will be Brent's one.

The exit condition and the convergence criterion of the optimizer depend on its intrinsic characteristics [43]. For this problem, they have been settled as follows:

- xtol= $1e^{-8} \longrightarrow$ relative error in the optimal solution acceptable for convergence.
- maxiter= 500 $\longrightarrow$ maximum number of iterations to perform.

The optimization over w was performed both using Brent's algorithm and the L-BFGS-B algorithm. Both methods lead to the optimal result, but Brent's algorithm converges much faster, hence being the preferable method. The time to reach convergence was used as the criterion for establishing the optimizer type.

5 Experimental results and discussion

The input of the DQC is a regular equidistant grid of $N = 200$ sampling points: 20 points for the spatial variable (x) and 10 points for the temporal (t). The spatial variable runs in the interval $0 \leq x < 1$ and the temporal variable runs in the interval $0 \leq t \leq 0.1$.

The artificial data is obtained by solving the Heat equation 3 for $w = 1$, i.e. $u_t - u_{xx} = 0$. This equation, together with the initial boundary value problem, is solved using the *Matlab* PDE solver named *pdepe* [44]. This solver solves initial-boundary value problems for systems of PDEs in one spatial variable x and time t. *pdepe* requires at least one parabolic equation in the system, so it is valid for solving the Heat equation. The *Matlab* code for obtaining the real data $u(x, t)$ is attached in Annex 9.3.

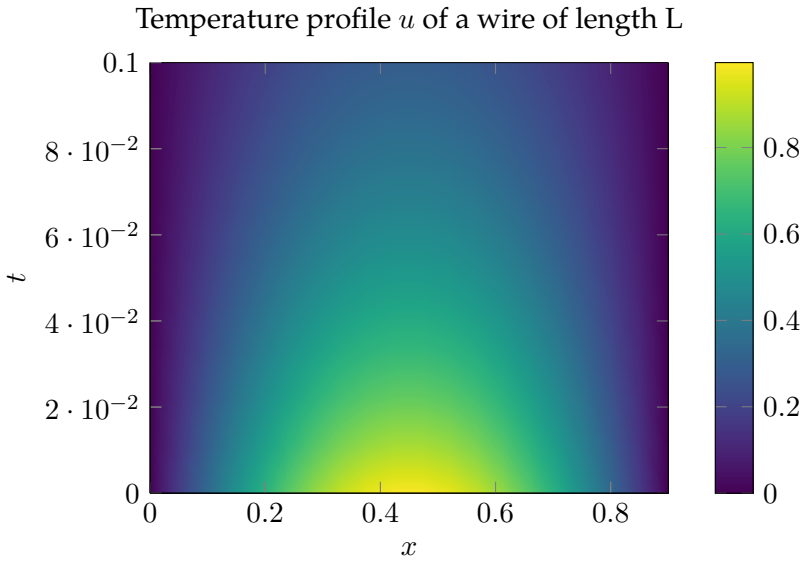

Figure 6: Real solution of the Heat equation with coefficient $w = 1$

Figure 6 shows the temperature distribution of a wire of length $L = 1$. According to the IBVP, the extremes of the wire are kept at a fixed temperature of $u_L = 0$. At the start of the simulation $(t = 0)$ the centre of the wire is heated up to $u_0 = 1$. At this point, the wire reaches its maximum temperature. As time goes by, one expects the heat to diffuse or be lost to the environment until the temperature of the bar is in equilibrium with the air $(u \to 0)$.

According to the second law of thermodynamics, heat flows from hotter bodies to adjacent colder bodies, in proportion to the difference of temperature and of the thermal conductivity of the material between them. So one should expect the temperature profile to decrease with time until reaching the steady state.

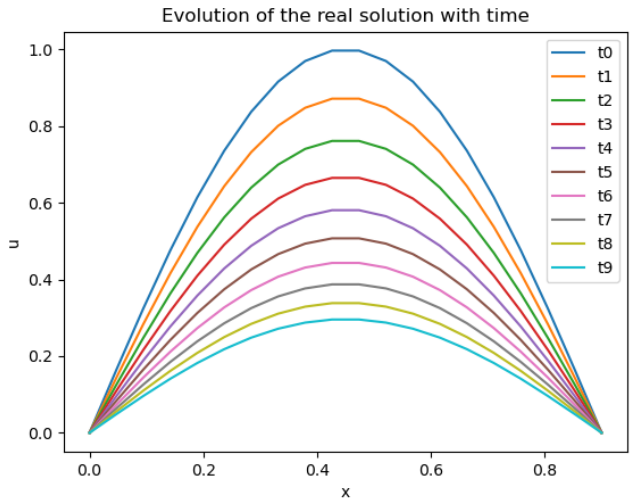

Figure 7: Evolution of the temperature profile over time.

In fact, the solutions of the heat equation are characterized by a gradual smoothing of the initial temperature distribution by the flow of heat from warmer to colder areas of the wire. This behaviour is exactly represented in figure 6: the temperature shows a very steep and clear parabolic profile at initial times that gradually dissipates into an almost flat line. In Figure 7 one can see that after 0.1s, the system has hardly reached the stationary state (since the temperature is not homogeneous all through the wire), but it has barely entered it, and so one can already guess the tendency that will follow the temperature in a near future (eventually reaching a flat line).

5.1 Workflow

For the experimental part, a code was programmed using *Python* that implemented the hybrid quantum-classical machine learning problem. The original code can be found in Annex 9.4 and it was built from scratch. In fact, there is some software that facilitates the implementation and programming of any quantum circuit using quantum gates (such as Qiskit, PennyLane and Cirq amongst others). However, the present code does not use any of them. The main reason for creating all the matrix and gate operations from scratch is the need to get acquainted with the mathematical part of the quantum mechanics behind the gate operations, which was believed to be useful from a mathematical understanding point of view. To this extent, this present section will describe the workflow followed by the code.

First the input for the solver is specified: a set of points X for each equation variable (t, x), comprised in a regular equidistant grid, and the grid is initialized. The set of boundary points X_{BC} is also extracted from the data-set X.

Next, the derivative quantum circuit is built:

(a) The type of quantum feature map and the nonlinear function define the encoding for a single variable. This encoding is the same for both the temporal $\hat{\mathcal{U}}_{\varphi_t}(t)$ and the spatial $\hat{\mathcal{U}}_{\varphi_x}(x)$ independent variables. Then, the full encoding $(\hat{\mathcal{U}}_{\varphi}(t, x)$ is built stacking the single encoders in parallel or in series. As discussed in section 4.2.3, the chosen stacking is the

parallel distribution.

(b) The ansatz of variational quantum circuit $\hat{\mathcal{U}}_\theta$ depends on the variational parameters θ and the number of layers according to the depth d.

(c) The cost operator is chosen as a total magnetization in the Z direction $\hat{C} = \sum_{j=1}^{n} \hat{Z}_j$.

(d) The type of loss function $\mathcal{L}_\theta$ and the strategy to match the boundary terms are defined according to the error distance strategy (L) and the weights of each loss term (λ). The loss function $\mathcal{L}_w$ accounting for the data fitting is also established.

(e) The classical optimizers are specified: L-BFGS-B for the variational angles (θ) and Brent for the unknown parameter (w) of the PDE.

The variational parameters are set to initial random values $\theta_0 \in [0, 2\pi]$. The expectation value over the variational quantum state $|u_{\varphi,\theta}(t,x)\rangle$ for the cost function is estimated using the quantum hardware for a chosen point $X_i = (t_i, x_i)$. Then the output solution $\hat{u}(t_i, x_i, \theta)$ at this point is constructed. Apart from the value of the function $\hat{u}$, the derivatives $\hat{u_t}$, $\hat{u_x}$ and $\hat{u_{xx}}$ are also calculated for that particular point using the parameter shift rule and different circuit evaluations.

The procedure of calculating $\hat{u}$, $\hat{u_t}$, $\hat{u_x}$ and $\hat{u_{xx}}$ is repeated for all X_i in X and also for all X_i in X_{BC}. All these values are collected and the loss function $\mathcal{L}_\theta$ can be composed for the entire grid.

At this point, the optimization strategy is performed in two different stages:

- Stage 1: The goal of the loss function $\mathcal{L}_\theta$ is to quantify how well the potential solution $\hat{u}$ (parametrized by the variational angles θ) satisfies the differential equation (parametrized by w). So the aim is to minimize the loss function $\mathcal{L}_\theta$. In order to do so, the optimizer L-BFGS-B computes the gradient of the loss function with respect to variational parameters θ (using a classical optimization procedure) and updates the variational angles at every iteration. The previous steps are repeated until the loss function value converges. After exiting the classical loop, the solution is chosen as a circuit with angles θ^* that minimize the loss. Finally, the full solution $u^*(\theta^*)$ is extracted by sampling the cost function for optimal angles $\langle u_{\varphi,\theta^*}(t,x)|\, \hat{\mathcal{C}}\, |u_{\varphi,\theta^*}(t,x)\rangle$.

- Stage 2: Once the optimal solution $u^*(\theta^*)$ is obtained, the goal of the loss $\mathcal{L}_w$ is to quantify how well the optimal solution for those θ^* fits the empirical data-set u (obtained through the *Matlab* solver). In order to minimize $\mathcal{L}_w$, the optimizer Brent computes the gradient of the loss function with respect to w and updates the value of w at every iteration. This step is repeated until the loss function value converges. After exiting the classical loop, the value that minimizes the loss function is w^* and this is given as a parameter to $\mathcal{L}_\theta$, going back to Stage 1. The goal is now to find the new θ^* that make the surrogate $\hat{u}$ satisfy the PDE parameterized by the new w^*.

This optimization in two steps is repeated until eventually both loss functions converge, leading to the final surrogate $\hat{u}$ and the inferred parameter $\hat{w}$ that was initially unknown.

5.1.1 Benchmarking

During the programming of code for the classical-quantum approach, several intermediate versions were developed and eventually discarded, until reaching the actual and optimized version presented in Annex 9.4.

The code adapted and evolved as different modifications were added. Although only the latest version is available here, several sanity checks were performed throughout the process in order to guarantee that the code did not contain errors. For instance, during the computation of the derivatives it was checked that the derivatives calculated through the parameter shift rule matched the ones calculated using finite differences.

Let $\hat{u}(t,x)$ denote the surrogate of the DQC and $\hat{u}_t(t,x)$, $\hat{u}_x(t,x)$ and $\hat{u}_{xx}(t,x)$ the corresponding derivatives of the output calculated with the quantum circuit through the parameter shift rule. The derivatives using finite differences were computed as follows:

$$u_t^{FD} = \frac{\hat{u}(t+h,x) - \hat{u}(t,x)}{h},$$

$$u_x^{FD} = \frac{\hat{u}(t,x+h) - \hat{u}(t,x)}{h},$$

$$u_{xx}^{FD} = \frac{\hat{u}(t,x+h) + \hat{u}(t,x-h) - 2\hat{u}(t,x)}{h^2}.$$

Since the goal is to check the implementation of the parameter shift rule and the performance of the DQC rather than its expressivity, it is sufficient to conduct this check with an arbitrary number of qubits, layers and random variational parameters (e.g. $n = 4$ and $d = 5$). The step for the numerical differentiation was chosen to be $h = 10^{-5}$. Figures 8, 9 and 10 show the relative error of the first and second temporal and spatial derivatives between the values computed via the parameter shift rule and the finite differences. In all this cases, the relative error at every point of the grid is smaller than 0.2%. This implies that the parameter shift rule is an accurate method for computing derivatives. The relative error function follows a completely random behaviour, with some points showing more error than others. There is not a reasonable explanation for this behaviour, since at each simulation the points with more error were different. Furthermore, it is not relevant to study the error difference between points, since it is of the order of 10^{-3}.

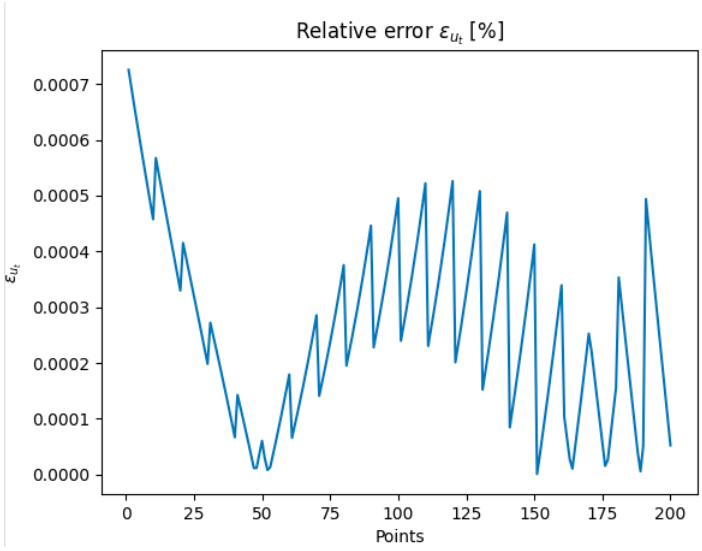

Figure 8: Relative error of the 1st time derivative $\epsilon_{u_t} = \frac{|u_t^{FD} - \hat{u}_t|}{u_t^{FD}} \cdot 100$

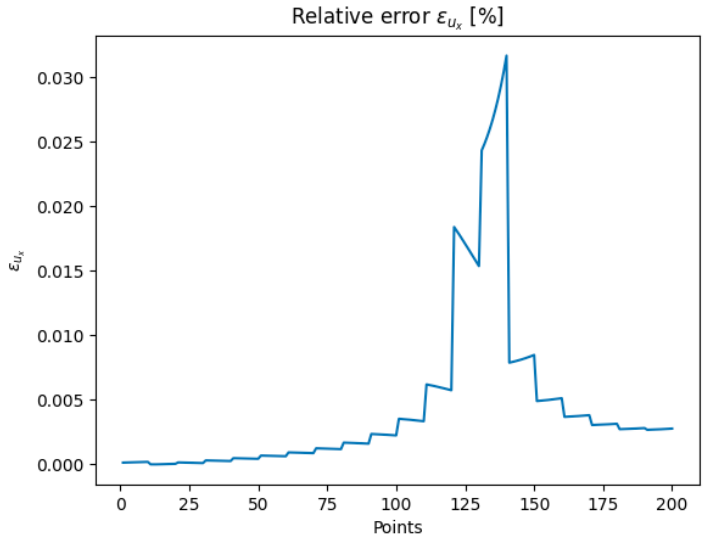

Figure 9: Relative error of the 1st spatial derivative $\epsilon_{u_x} = \frac{|u_x^{FD} - \hat{u}_x|}{u_x^{FD}} \cdot 100$

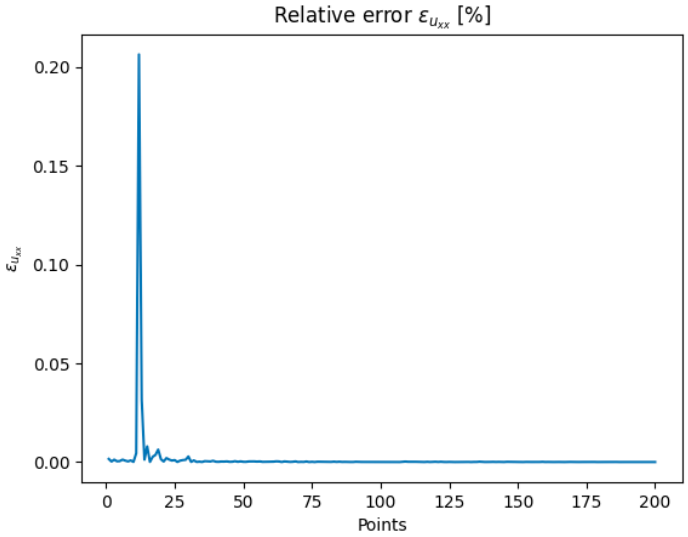

Figure 10: Relative error of the 2nd spatial derivative $\epsilon_{u_{xx}} = \frac{|u_{xx}^{FD} - \hat{u}_{xx}|}{u_{xx}^{FD}} \cdot 100$

Another sanity check was to analyse the behaviour of the loss function $\mathcal{L}_\theta$ in terms of the values of the variational parameters θ. Since θ correspond to rotational angles, they should present a 2π-periodicity. For a given rotation angle θ_1, when plotting the loss function for different values of θ_1 one should expect to obtain a sinusoidal behaviour. Indeed, figure 11 exhibits this behaviour. Furthermore, the periodicity appears every 2π, independently of the parameter θ_i chosen.

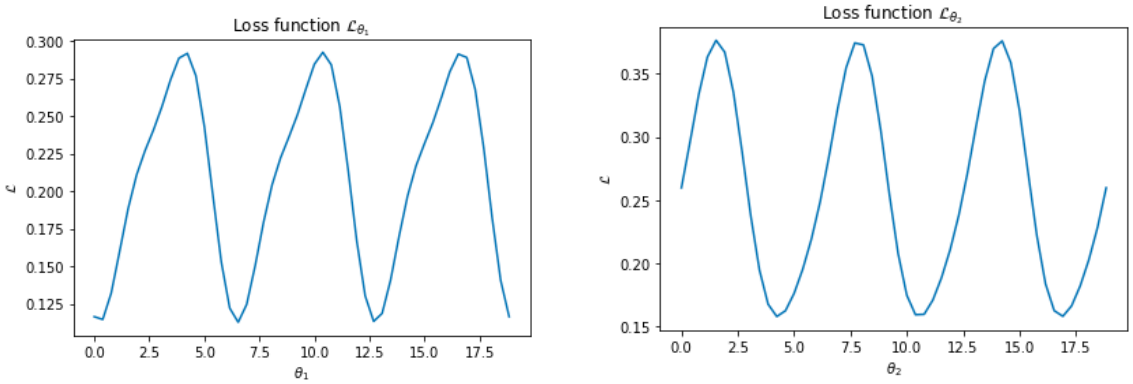

Figure 11: Loss function $\mathcal{L}_\theta$ for different values of θ_1 (left) and θ_2 (right)

Also, different trials of the simulations were done changing the quantum feature map. The product map $\varphi_j(x) = j \arccos x$ did not offer good results, since the accuracy of the surrogate came to be very weak in comparison with $\varphi_j(x) = 2j \arccos x$. The results were discarded, so they are not included in the thesis. However, they provided further proof that the choice of the nonlinear encoding function can indeed change the expressivity of the circuit, as discussed in section 4.2, and that the factor 2 plays an important role in the expressive power of the DQC.

Some other simulations were done using the matrix decomposition into single qubits Pauli gates, as described in section 4.2.2. This approach is quite useful when dealing with more

qubits, since the size of the matrices scales with up to 2^n. The simulations performed in this thesis were limited to at most $n = 6$ qubits. Thus for this amount of qubits, the usual matrix multiplication method was found to be faster than the matrix decomposition method. With the only purpose of reducing the computational time, all the further simulations will be performed using the usual $2^n \times 2^n$ matrix multiplication. However, whenever increasing the number of qubits significantly, the approach of decomposition should also be considered and, if possible, prioritized, since it can achieve an important speedup.

5.2 Expressivity of the quantum circuit

The accuracy of the surrogate $\hat{u}$ obtained by the hybrid combination DQC (quantum) - optimizer (classical) depends on several variables that can be adjusted. This first section takes a look into the impact of these variables upon the surrogate $\hat{u}$ and the effect of changing the magnitude of these variables. There are four "control variables" that can be adjusted: (1) number of qubits n, (2) depth or number of layers of the VQA d, (3) the error distance function L that quantifies the loss function, (4) the weights λ of the loss function. The number of qubits and of layers act on the DQC (quantum hardware), and the error types and the weights act on the optimizer (classical hardware).

In this section the DQC will be used as a solver for the Heat equation with $w = 1$ in order to obtain a surrogate of the solution. This surrogate $\hat{u}$ will be analyzed and compared to the real one u obtained via the *Matlab* solver. Several trials will be performed, each changing the "control variables". The values that will be considered for each variable are the following:

1. Number of qubits (n): 2, 4, 6. It is important to note that, due to the implementation of a parallel feature map (as discussed in section 4.2) the number of qubits of the circuit must be even.

2. Number of layers (d): 5, 10, 20

3. Error distance (L): MSE, MAE, MAX

4. Weights: Λ_1, Λ_2

   In order to calculate the numerical values of every weight, it is necessary to know the number of points of every boundary. For this particular case: $N = 200$, $M_{X_l} = 10$, $M_{X_r} = 10$, $M_{T_0} = 18$. Applying the mathematical expressions described in Table 4, one obtains the following values, gathered in Table 5.

Weights	Λ_1	Λ_2
λ_f	0.0192	0.84
λ_{bl}	0.384	0.042
λ_{br}	0.384	0.042
λ_{b0}	0.213	0.0756
$\sum_i \lambda_i$	1	1

Table 5: Numerical values of the weights of each loss term.

The effect of changing each variable is resumed in Table 6, which includes the values of each loss function term after the optimization process has converged.

n	**d**	**L**	Λ	$\mathcal{L}$	$\mathcal{L}_f$	$\mathcal{L}_{X_l}$	$\mathcal{L}_{X_r}$	$\mathcal{L}_{T_0}$
2	5	MSE	1	0.0617	0.017	0.00639	0.00354	0.0411
2	10	MSE	1	0.0721	0.00543	0.0122	0.00269	0.0518
4	5	MSE	1	0.00760	0.00261	0.000444	0.000821	0.00373
4	10	MSE	1	0.00733	0.00247	0.000486	0.000837	0.00354
4	10	MSE	2	0.00783	0.000850	0.000609	0.000854	0.00551
4	10	MAX	1	0.0132	0.00565	0.000258	0.000561	0.00672
4	10	MAX	2	0.0146	0.00183	0.00151	0.00212	0.00910
4	10	MAE	1	0.0398	0.0123	0.0000567	0.000541	0.0269
4	10	MAE	2	0.0274	0.00126	0.00261	0.00332	0.0202
4	20	MSE	1	0.00757	0.00261	0.000448	0.000965	0.00354
4	20	MSE	2	0.00873	0.000778	0.00221	0.00096	0.00479
6	5	MSE	1	0.00485	0.00197	0.000206	0.000630	0.00205

Table 6: Values of the loss function terms for every combination of "control variables"

Let's analyse the effect of each "control variable" independently.

5.2.1 Number of qubits

From Table 6, one can see that the combination of variables that exhibits the least total loss function corresponds to $n = 6$ qubits. Indeed, it makes sense that increasing the number of qubits will translate into an increase in the accuracy and expressive power of the surrogate calculated by the DQC. However, the increase in qubits also increases the computational cost and time. In fact, simulations for $n = 6$ qubits took over 10 hours, while simulations for $n = 4$ qubits took a little more than 1 hour. So, there exists a compromise between efficiency (related to time consuming resources) and accuracy (related to the total loss function). For the same depth, error distance and λ configuration, increasing the number of qubits from $n = 6$ to $n = 4$ decreases the loss function by a 0.6 factor (increasing the accuracy) but increases the time consumption by a factor of 10 (decreasing the efficiency). Hence, since using $n = 4$ still offers accurate results, the higher order of qubits will be discarded. The next combination that offers the lowest loss function corresponds to: $n = 4, d = 10, L = MSE$. As an initial hypothesis, this will be considered the best option for the circuit variables.

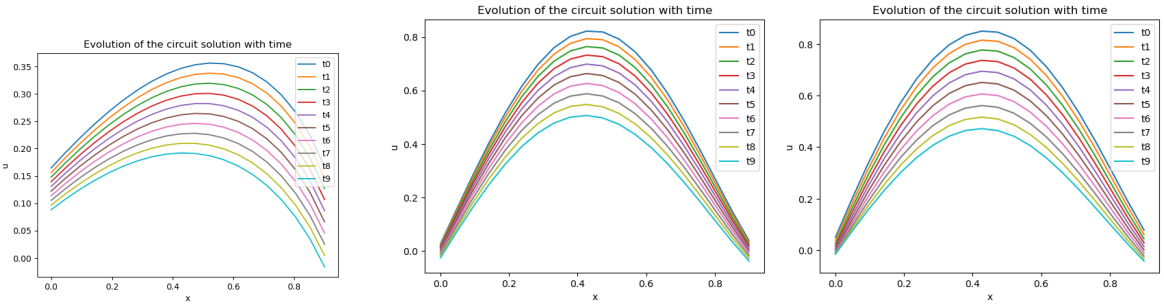

Figure 12: Evolution of the temperature profile over time for n=2, n=4, n=6

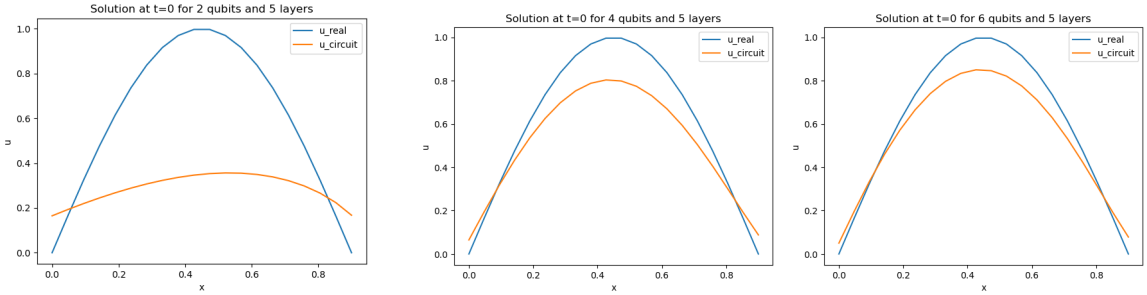

Figure 13: Temperature profile at t=0 for n=2, n=4, n=6

Keeping the other "control variables" constant at the optimal configuration ($d = 5$, $L = MSE$), figures 12 and 13 show that for $n = 2$ the DQC is not able to recover the original behaviour of the solution u. However, the behaviour of the surrogate $\hat{u}$ does not change significantly when increasing the number of qubits from $n = 4$ to $n = 6$.

5.2.2 Number of layers

The control variables will be fixed at $n = 4$, $L = MSE$.

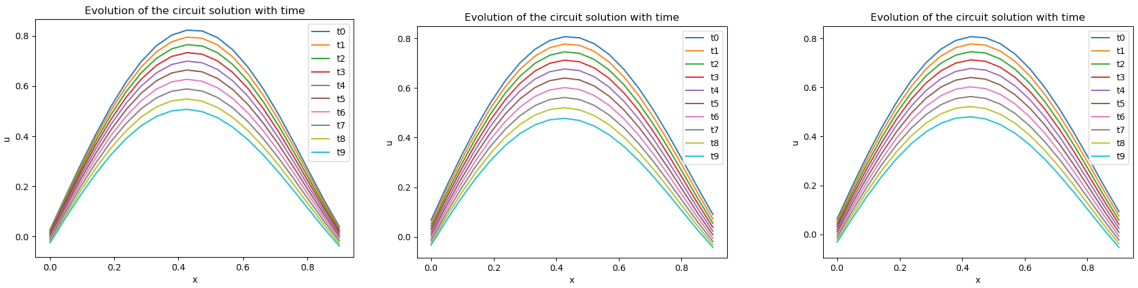

Figure 14: Evolution of the temperature profile over time for d=5, d=10, d=20

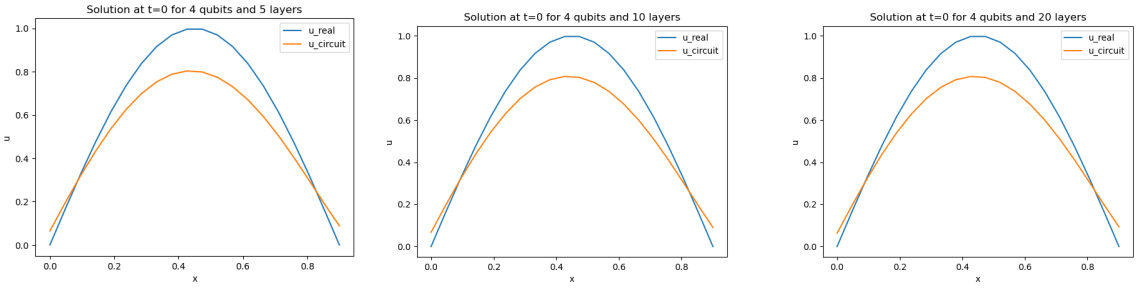

Figure 15: Temperature profile at t=0 for d=5, d=10, d=20

When taking into account time-computing resources, for lower depths the solver is slower to converge and does not reach as high accuracy as it does for higher depths. As depth increases more layers of parametrized gates are included in the variational ansatz and so the number of variational angle parameters increase. This causes an increase in the number of gate operations

needed in each iteration and how many parameters the classical optimizer needs to update, raising the time taken per iteration. Hence the lower and higher depths will be discarded, keeping an intermediate value of $d = 10$. Nevertheless, figures 14 and 15 show that the behaviour of the surrogate $\hat{u}$ does not change significantly when changing the depth of the VQA.

5.2.3 Error distance and weights

As previously discussed, the variables that obtain the optimal results correspond to $n = 4$, $d = 10$.

The first configuration of weights emphasizes the boundary terms.

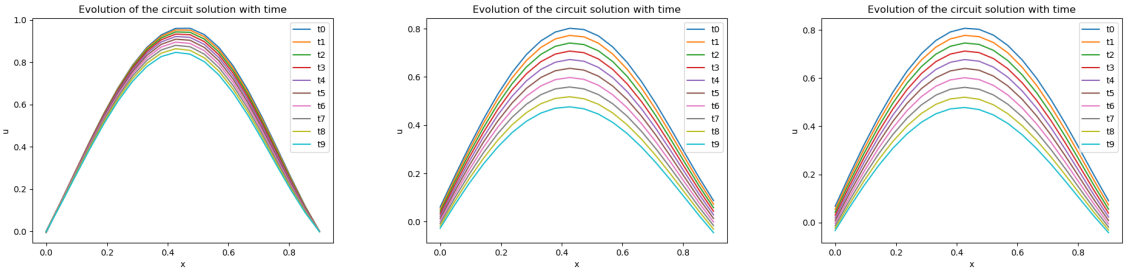

Figure 16: Evolution of the temperature profile over time for L=MAE, L=MAX, L=MSE and Λ_1

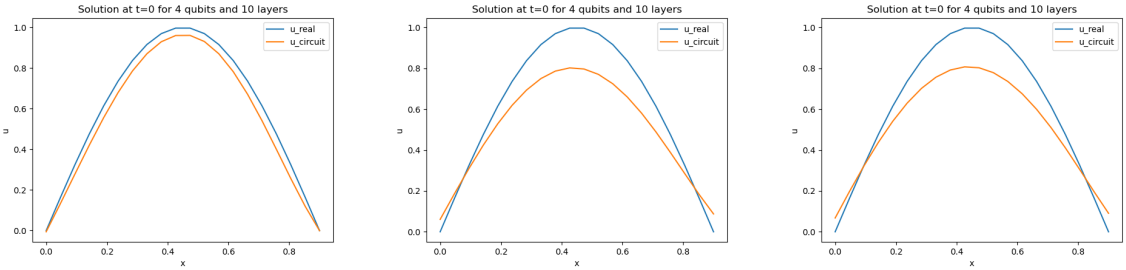

Figure 17: Temperature profile at t=0 for L=MAE, L=MAX, L=MSE and Λ_1

The second configuration of weights emphasizes the interior points of the domain.

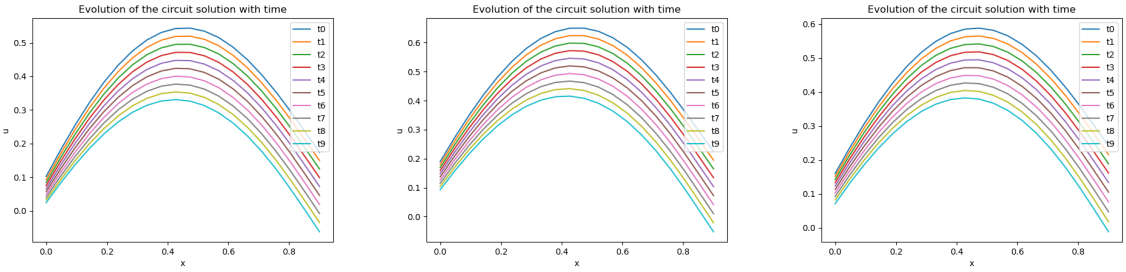

Figure 18: Evolution of the temperature profile over time for L=MAE, L=MAX, L=MSE and Λ_2

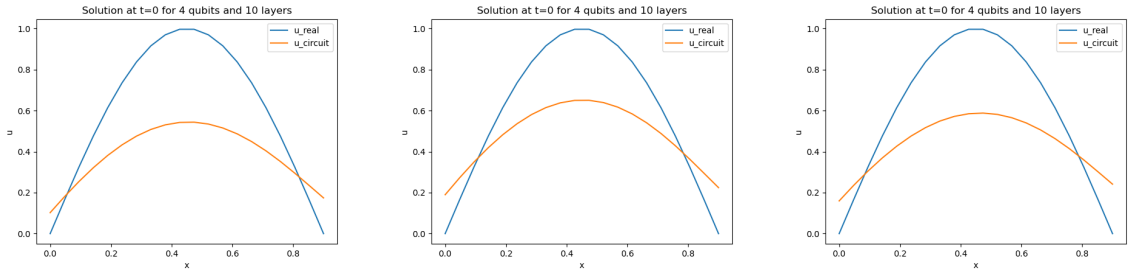

Figure 19: Temperature profile at t=0 for L=MAE, L=MAX, L=MSE and Λ_2

Independently of the weights' configuration, the error distances MAE and MAX are more accurate at the boundary points. However, the MSE gives a better expression of the surrogate of the solution at the interior domain. In fact, according to Table 6, the more accurate combination with a lower loss function corresponds to the MSE treatment of the error.

For the final configuration $n = 4$, $d = 10$ and $L = MSE$, the surrogate temperature profile of the wire can be plotted in Figure 20. In both cases, the diffusive behaviour of the heat equation can be appreciated.

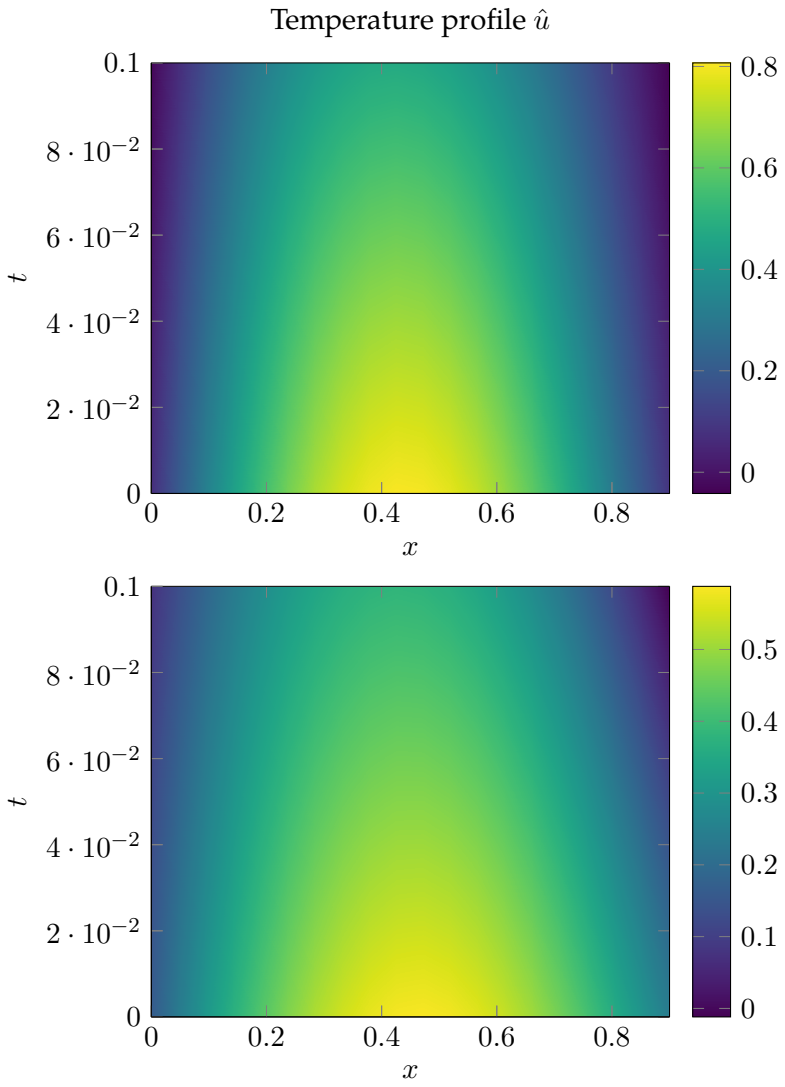

Figure 20: Surrogate solution of the Heat equation with $w = 1$ for Λ_1 (up) and Λ_2 (down)

Comparing the surrogate temperature with the real temperature map leads to Figures 21 and 22.

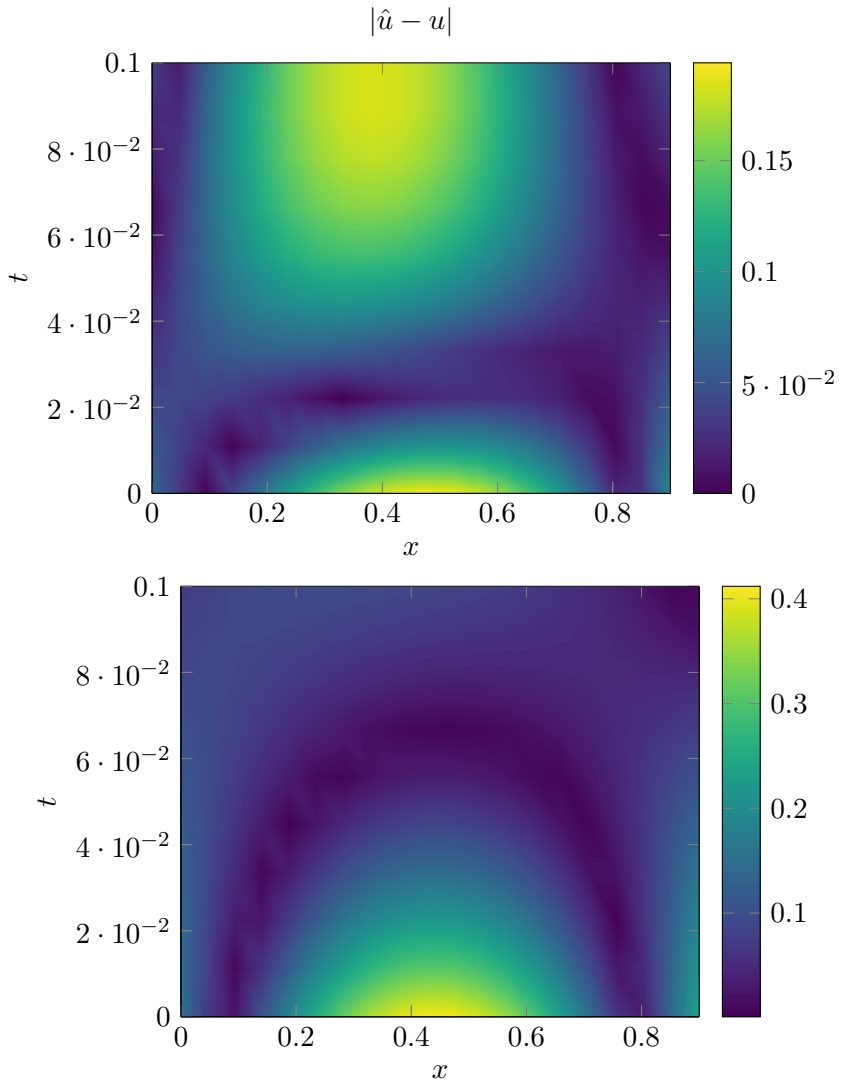

Figure 21: Absolute error for Λ_1 (up) and Λ_2 (down)

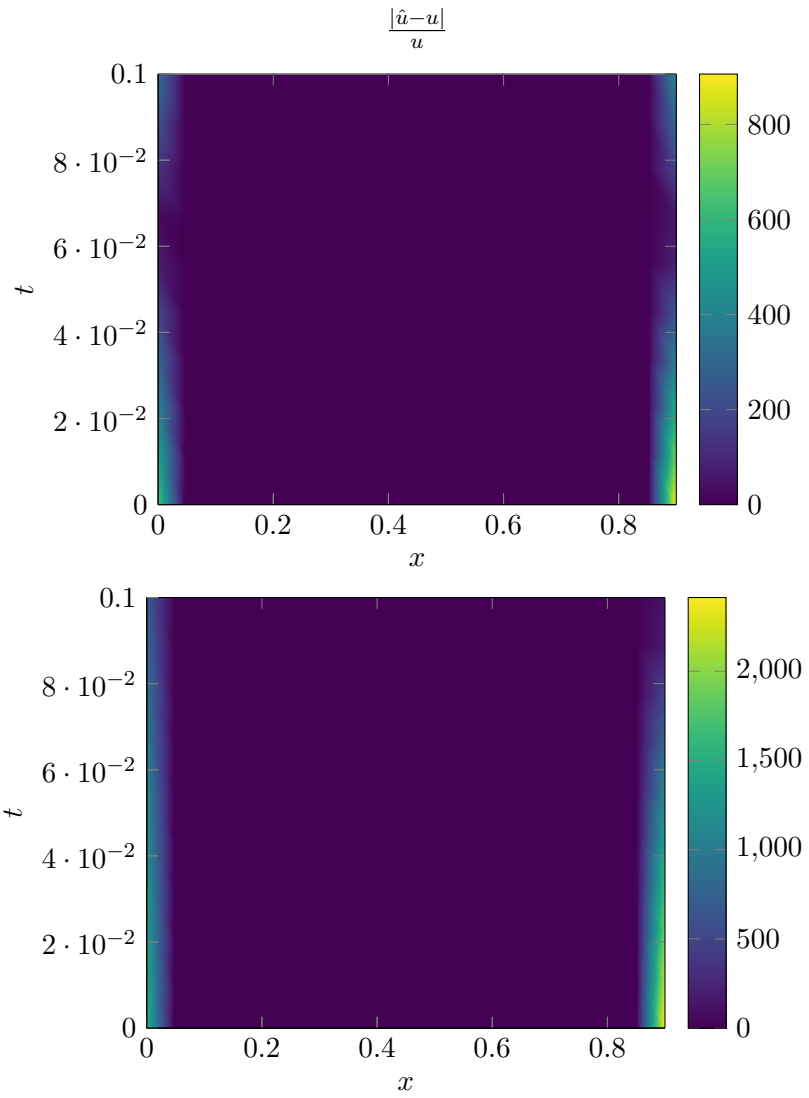

Figure 22: Relative error for Λ_1 (up) and Λ_2 (down)

Since Λ_1 gives more importance to the boundaries, its maximum error occurs in the interior domain with a value of 0.2. On the other hand, since Λ_2 gives more importance to the interior points, its maximum error occurs at the boundaries (particularly, for the initial time $t = 0$) with a value of 0.42.

The training of the L-BFGS-B optimizer for the θ parameters is shown in Figure 23. It can be seen that the loss function converges to 0 within 4000 iterations.

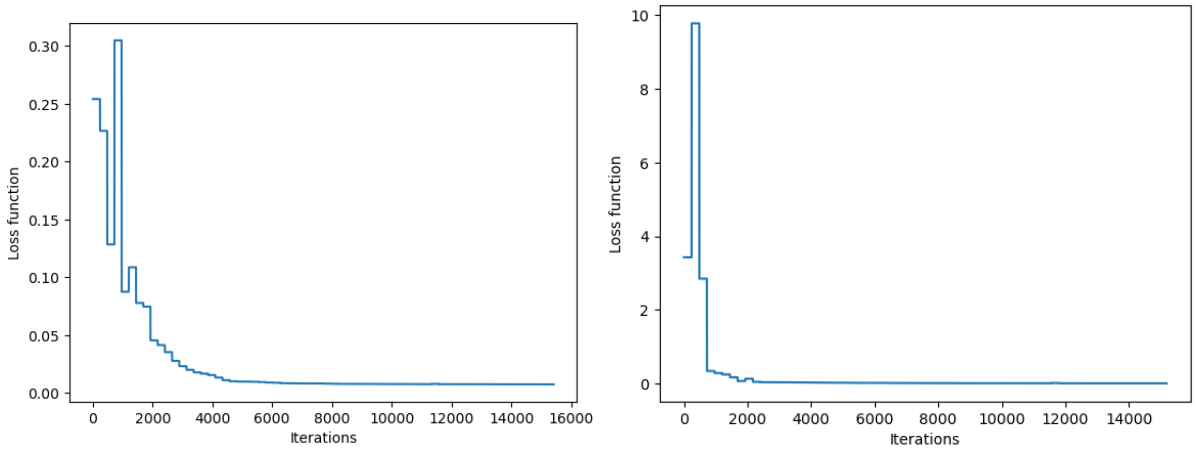

Figure 23: Convergence of the loss function $\mathcal{L}(\theta)$ for Λ_1 (left) and Λ_2 (right)

5.3 Parameter inferring

Once the control variables have been analysed, one can establish that the setup that gives the more accurate result corresponds to a DQC with $n = 4$ qubits, $d = 10$ layers and $L = MSE$ as loss function. With this setting, a quantum neural network will be used as the UFA to approximate $\hat{u}$ and perform parameter inference.

For the parameter inference, the simulations were performed under two different conditions: (1) non-noisy data, (2) data with 5% of Gaussian noise. The results obtained after the simulations are summed up in Table 7.

w^*	Λ	**Noise level** $[\%]$	ϵ_{rel} $[\%]$
1.14	1	0	14
1.11	2	0	11
1.31	1	5	31
1.27	2	5	27

Table 7: Estimation of the coefficient w for noisy and non-noisy data

The relative error in Table 7 is used as a measure to compare the PDE parameter $\hat{w}$ estimated by the DQC with the true coefficient $w = 1$ that generated the data (called truth). Even under the effects of noise, the DQC is able to estimate the unknown parameter with a relative error of 27%.

In order to do the optimization over w, it is very important for the loss function $\mathcal{L}(w)$ to be convex, i.e. to have only one minimum. Otherwise, the optimizer could get trapped into a local minimum. Figures 24, 25 and 26 show that the loss function, no matter the noise level and the Λ configuration, is convex in the domain of interest and has a single minimum, which corresponds to the optimal value w^* of the parameter w to be estimated. Once it has been proven that the loss function is convex, the training can be done.

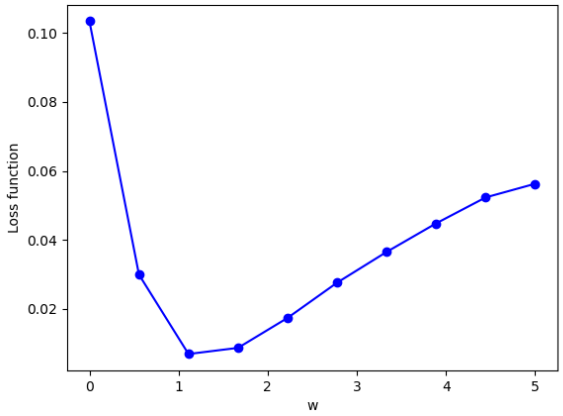

Figure 24: Loss function $\mathcal{L}(w)$ with non-noisy data and Λ_2

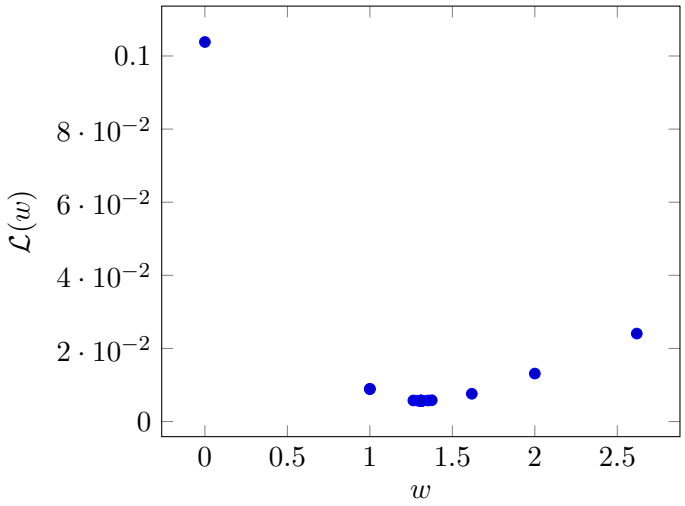

Figure 25: Loss function $\mathcal{L}(w)$ with 5% noisy data and Λ_1

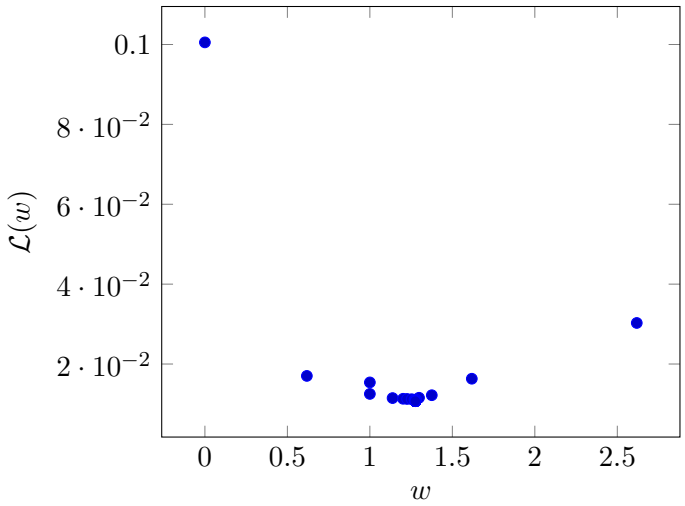

Figure 26: Loss function $\mathcal{L}(w)$ with 5% noisy data and Λ_2

The training of the Brent optimizer for the w parameter is shown in Figure 27. It can be seen that the loss function converges to 0 with 10 iterations.

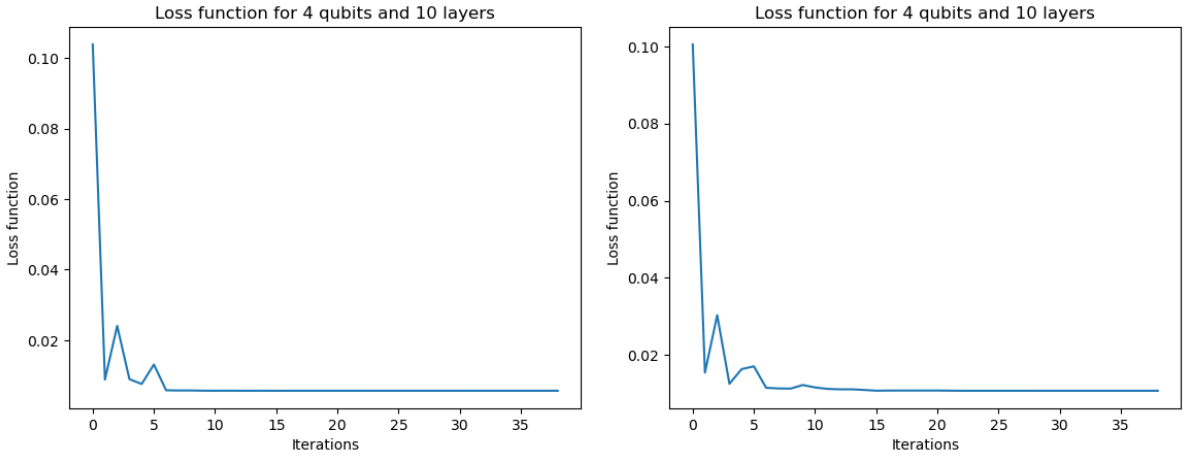

Figure 27: Convergence of the loss function $\mathcal{L}(w)$ for Λ_1 (left) and Λ_2 (right)

Both for noisy and non-noisy data, the most accurate results for w correspond to the setting Λ_2, i.e. for the configuration that gives more weight to the data-set and less to the boundary conditions. For the case of the Heat equation, the boundary conditions were not so relevant. It could be, though, that some other PDE has important conditions to be fulfilled at the boundary. In this case, the optimal setting would be to use the Λ_1 configuration.

6 Conclusions

This work has presented an hybrid quantum-classical approach for inferring unknown parameters of PDEs.

- The quantum part makes use of a differentiable quantum circuit on gate-based quantum hardware in order to solve the PDE. The method makes use of quantum feature map circuits to encode function values into a latent space. This allows to consider spectral decompositions of the trial solutions to differential equations. As shown in section 5.2, the method proposed in this thesis can accurately represent solutions using the high-dimensional Hilbert space of a qubit register thanks to the large spectral basis set of Chebyshev polynomials. The project also shows how analytical circuit differentiation via the parameter shift rule can be used to represent the function derivatives that appear in the PDE of interest, and how to construct loss functions whose aim is to improve the prepared trial solution.

- The classical part performs the optimization technique that allows to update the parameters that need to be trained: both the internal variational θ parameters and the ones that need to be estimated w.

This hybrid strategy has been able to correctly infer the unknown coefficients of a PDE even when the empirical observations are under the effect of Gaussian noise. As an example, the project presents solutions and parameter inference of the Heat equation.

It is important to note that the experiments performed here involve classical simulators of quantum computers with unrealistic assumptions such as zero sampling noise, no environment noise and no measurement noise. Some simulations were made including Gaussian noise in the experimental data, but the other sources of noise are still to be considered, since the effect of noise could highly affect the accuracy of the VQA.

This thesis has also performed an exhaustive analysis of the treatment and consideration of boundary and initial conditions. This study allows to conclude to what extent the boundary and initial conditions can affect the dynamics of a system.

This project has implemented a solver and an automated parameter inference approach using a differentiable quantum circuit strategy. In particular, the numerical experiments in section 5 show that relatively small quantum circuits can be used for meaningful computations in scientific machine learning. Thus, these results are a vivid example that the fields of quantum computing and scientific machine learning are moving closer together.

6.1 Bottlenecks and limitations

Due to the high time-consuming circuit, the simulations were performed using at most 6 qubits. It would be interesting to investigate the performance of the circuit for a larger number of qubits. Nevertheless, increasing the number of qubits can lead to the *barren plateaus*[15] phenomenon, which directly impacts on the trainability of the circuit. When a loss function exhibits a barren plateau (BP), it can have gradients that vanish exponentially in the number of qubits n. That

[15] Areas in the cost landscape where the gradient of a parameterized circuit becomes negligible.

is, the magnitude of its partial derivatives will decay exponentially with the system size [2], meaning that the deeper the circuit, the more it will be affected. Since gradients of the loss function are crucial in the optimization procedure, analyzing the existence of BPs in a given VQA is fundamental to preserve the quantum advantage achieved with the PQC.

Since deep PQCs exhibit BPs when randomly initialized [2], the probability of finding the solution when randomly initializing the ansatz is exponentially small. There are two approaches for avoiding or mitigating the effects of BPs:

- Parameter initialization: randomly initializing an ansatz can lead to the algorithm starting far from the solution, near a local minimum, or even in a region with barren plateaus. Hence, optimally choosing the seed for θ at the beginning of the optimization is an important task. There exist several strategies for wisely initializing θ_0 [2]. In this work, the randomization of the parameters was restricted in the interval $[0, 2\pi]$, because the parameters to be optimized correspond to angles; hence they have a 2π-periodicity.

- Ansatz strategies: another strategy for preventing BPs is using particular structured ansatzes in order to restrict the space explored by the ansatz during the optimization procedure. As the depth of the ansatz continues to increase, eventually the problem of barren plateaus can be encountered. However, there are certain architectures of QNN that are immune to barren plateaus, and hence are trainable even for large problems.

In order to find the optimal circuit angles θ^*, the gradient of the loss function w.r.t the parameters θ is computed by the L-BFGS-B optimizer via finite-differences. The vanishing gradients caused by BPs can cause the solver to struggle to improve the parameters. Hence, BPs will impact on optimization strategies that go beyond (first-order) gradient descent, which is the case of the L-BFGS-B solver. The solution to this problem comes with changing the approach for calculating the gradients w.r.t. θ. For instance, analytically evaluating any high-order partial derivative on the DQC using automatic differentiation through the parameter shift rule in an approach known as *stochastic gradient descent* [45] [46]. This implies taking advantage of the DQC not only to compute the derivatives w.r.t. the inputs t and x, but also w.r.t. θ. This approach is one of the most common in optimization, and it is based on making iterative steps in directions indicated by the gradient. Given that only statistical estimates are available for these gradients, these strategies are known as stochastic gradient descent (SGD) methods. One SGD method that is widely used in machine learning is Adam [2], which adapts the size of the steps taken during the optimization to allow for more efficient and precise solutions than those obtained through basic SGD.

Using the chain rule, the derivative $\frac{\partial \mathcal{L}}{\partial \theta_j}$ can be written as a function of the derivatives of the variational circuit $\nabla_\theta \hat{\mathcal{U}}_\theta$, which are measured using the automatic differentiation approach (parameter shift rule), as presented in section 4.2.1. In particular, the work done by [45] shows an example of how to apply the chain rule to decompose the mean square error (MSE) loss function into the derivatives of the expectation values of the circuit output.

As analysed in section 4.1, circuit learning using analytical gradient outperforms any finite difference method. This is done by showing that for n qubits and precision ϵ, the query cost of an oracle for convex optimization in the vicinity of the optimum scales as $\mathcal{O}(\frac{n^2}{\epsilon})$ for the analytical gradient, whereas finite difference needs at least $\Omega(\frac{n^3}{\epsilon^2})$ calls to the oracle [1].

6.2 Time and economic cost

Regarding the time dedication, the effective starting day of the project was 1^{st} February 2022. The last working day was 31^{st} August 2022, the day on which the presentation of the thesis took place. This adds up to almost 30 weeks of full-time dedication. The daily working hours in the office add up to 7-8 h/day. Supposing a normal week of 5 working days, this implies a total of 35 to 40 h/week. In global, the whole thesis demanded more than 1000 hours of work.

The project has been divided into several tasks in order to facilitate the organization and assessment of the full problem.

1. **Task 1**: reading and collection of theoretical information about quantum algorithms applied to solving PDEs and choice of the research question.

2. **Task 2**: understanding of the theoretical background and implementation of parameterized quantum circuits and quantum machine learning.

3. **Task 3**: generation of usable experimental data (through *Matlab*) and code programming (with *Python*).

4. **Task 4**: application of modifications, improvements and new versions of the code. Code optimization and speed up.

5. **Task 5**: performing simulations and obtaining results.

6. **Task 6**: writing the thesis and preparation of the oral presentation.

Figure 28 shows the Gantt diagram for this project according to the tasks described above. Gantt diagrams are widely used in engineering when it comes to project management. They show a list of the different activities that need to be developed, their duration and the resources to be used next to a calendar-like table. Each task is represented with an horizontal bar based on its initial and ending date, covering the calendar time needed to complete each task. Gantt diagrams are useful when planning the development of projects, since they are very simple and easy to interpret. However, they do not allow to represent the dependencies between tasks. Furthermore, a Gantt diagram is not so accurate when performing a scientific project, because research is completely unpredictable and new. This novelty introduces a degree of uncertainty that translates into alterations of the initial plan and strategy and leads the researcher to constantly adapt to the arisen problems. Another characteristic of scientific research is the rate of completion of the stages of a project: since it implies a learning process, the later stages are usually completed faster than the initial ones, since the researcher obtains knowledge and becomes acquainted with the theory background and techniques of the studied subject. In the same way, the stages that involve research and novel topics that have not been discovered yet (tasks 3, 4, 5) do not promise an end date and can prolong further in time, so they usually take longer than those that are sure to be completed (tasks 1, 2, 6).

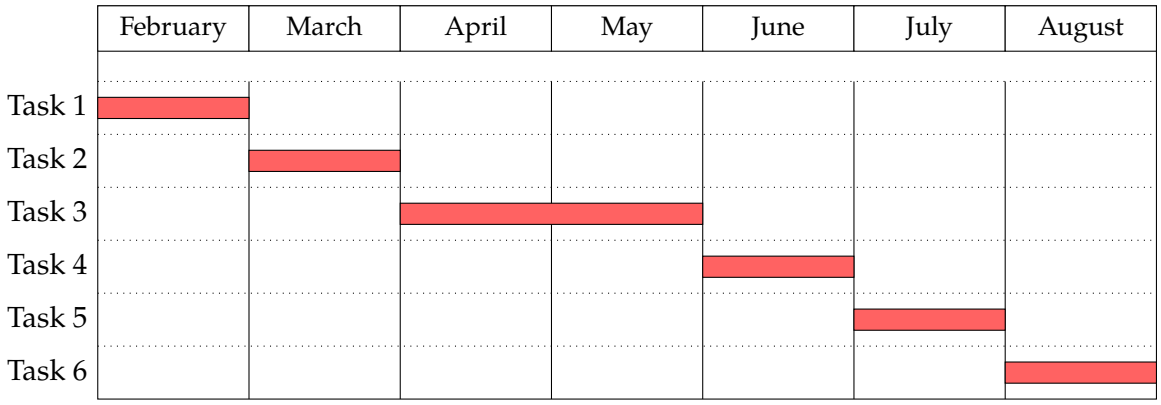

Figure 28: Gantt's diagram

Regarding the financial side of the project, it is quite hard to estimate the economic cost, since no additional resource was needed other than a computer to program the code. The computer also granted access to the cluster of the Lorentz Institute, which was very helpful in order to speed up the computations and simulations. In fact, the simulations were very time-consuming, ranging from 1 hour (for the simplest ones using only 2 qubits) to 10 hours (for the ones using 6 qubits). So, the only significant costs that will be considered are the ones associated with energy consumption: $0.05kWh$ of the researcher's personal working station $(8h/day)$, $8kWh$ for daily general energy consumption $(8h/day)$ and $0.4kWh$ the usage of the computer for accessing the cluster at the stages of the project involving simulations $(16h/day)$. For the sake of simplicity, energy usage will be charged a rate of $0.30€/kWh$. Hence, the cost of this project is depicted in table 8.

Concept	**€/week**	**Duration [weeks]**	Cost [€]
Personal workstation	0.6	30	18
Office energy consumption	96	30	2880
Cluster usage	9.6	4	38.4
Total cost			2936.4

Table 8: Economic cost of the project

7 Further work

This section describes some ideas that can be performed in a different way as further work.

7.1 Improvements

If one were to extend this research project, here are some ideas that can be derived from the work done in this thesis.

- **Change the encoding map to achieve better expressivity**: the feature map that encodes the input variables of the DQC can be changed from a Chebyshev basis to a Fourier feature map or other universal basis function sets. It would be interesting to analyse and compare the final output using a Fourier encoding map.

- **Include a regularization term in the loss function to achieve better expressivity**: as described in section 4.3, the regularization term helps the optimizer to avoid getting trapped in local minima, hence providing the actual solution and increasing the expressivity of the DQC.

- **Compare the quantum with the classical approach**: an important question to discuss is the potential and advantage achieved by quantum machine learning when it comes to solving differential equations with respect to classical methods. This project has shown that the representation power of quantum circuits grows with the number of qubits, and this leads to increased accuracy even for small networks. Hence, quantum circuits offer an expressivity advantage. However, it would be interesting to build a classical approach and compare the results obtained both with the DQC and the PINN architectures.

- **Extend to other types of PDEs**: This project has analysed a basic example of second order linear PDE (the Heat equation). It would be interesting to extend this quantum approach to other types of equations, such as nonlinear PDEs and systems of PDEs. For instance, nonlinear systems of PDEs are one of the most challenging equations to solve. As an example, the most known nonlinear PDE is Burger's equation, which occurs in various areas of gas dynamics, fluid mechanics and applied mathematics. Furthermore, it is evoked as a prime example to benchmark model discovery and coefficient inference algorithms [17], as it contains a non-linear term as well as second order spatial derivative.

  $$u_t = \nu u_{xx} - u u_x \tag{9}$$

  where ν is the viscosity of the fluid and u its velocity field.

- **Work with multiple dimensions**: this thesis has worked with 2 independent variables: the temporal (t) and a spatial (x). A further work could be to extend to d spatial dimensions. However, when increasing the number of variables, this implies dealing with the *curse of dimensionality*.

- **Infer more than one parameter**: the Heat equation has a single coefficient w. However, the parameter estimation can also be performed with more than one coefficient. In this case, the parameters $\mathbf{w}$ are treated as a vector (analogous to the case of θ). Then, they can no longer be optimized with a 1D optimizer. Instead, the L-BFGS-B optimizer can be used to infer the vector of unknowns.

- **Extend to model discovery**: a possible further work could be to extend DQCs to the discovery of unknown differential equations from empirical measurements. Model discovery (often referred to as equation discovery) tries to find a mathematical expression that describes a given spatio-temporal data-set.

  This approach, instead of solving a known PDE, consists on finding an unknown PDE only from measurements and a predetermined library of basis functions. The PDE underlying a data-set $u(t, x)$ is discovered by writing the model discovery task as a sparse regression problem,

  $$\partial_t u = \Theta \xi \tag{10}$$

  where $\partial_t u$ is a column vector of size N (number of points of the data-set) containing the time derivative of each sample. Θ is a matrix containing a library of polynomial and spatial derivative functions (for instance: u, u_x, u_{xx}, uu_x...) and can be written as:

  $$\Theta = \begin{bmatrix} 1 & u(t_0, x_0) & u_x(t_0, x_0) & ... & u^2 u_{xx}(t_0, x_0) \\ 1 & u(t_1, x_1) & u_x(t_1, x_1) & ... & u^2 u_{xx}(t_1, x_1) \\ ... & ... & ... & ... & ... \\ 1 & u(t_N, x_N) & u_x(t_N, x_N) & ... & u^2 u_{xx}(t_N, x_N) \end{bmatrix}$$

  Since Θ contains significantly more terms than required, most coefficients in ξ will be zero and hence model discovery turns into finding a sparse representation of the coefficient vector ξ.

  The problem is trained by optimizing a loss function that includes:

  - $\mathcal{L}_d$ accounts for the fitting of the empirical data-set, evaluated at the N points of the grid.

    $$\mathcal{L}_d(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(\hat{u}(t_i, x_i, \theta), u_i).$$

  - $\mathcal{L}_p$ an additional term accounting for the library of functions. Additionally, this term acts as a regularizer on $\hat{u}$, preventing overfitting of the noisy data-set.

    $$\mathcal{L}_p(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(\partial_t \hat{u}(t_i, x_i, \theta), \Theta_{ij} \xi_j).$$

  Model discovery has already been solved from a classical perspective [17], but it can be interesting to analyse how it performs in a quantum setup.

- **Use a real quantum computer to perform the simulations**: various strategies of error mitigation have been proposed that can further improve the performance of algorithms when run on physical devices. Although many error mitigation strategies for NISQ hardware exist, it should still be investigated how well these strategies work in more realistic settings and importantly on real quantum hardware, and which quantum hardware is most compatible or natural for this strategy.

7.2 Applications

The work presented in this thesis may be useful when dealing with stochastic differential equations (SDE), which emerge when dealing with Brownian motion and quantum noise [40]. A general system of stochastic differential equations can be written as

$$dX_t = f(X_t, t)dt + g(X_t, t)dW_t$$

where X_t is a vector of stochastic variables parameterized by time t (or other parameters). Deterministic functions f and g correspond to the drift and diffusion processes, respectively. W_t corresponds to the stochastic Wiener process (standard Brownian motion). The stochastic component makes SDEs distinct from other types of partial differential equations, adding a non-differentiable contribution. This also makes SDEs generally difficult to treat.

SDEs are widely used in financial calculus, since there is often not exact knowledge of the market dynamics but there is access to historical data. The parameter inference and model discovery could infer the SDE from data and make better predictions compared to a purely data-only approach. Real-world applications of SDEs in finance lie in predicting stock prices or currency exchange rates.

The heat equation can be used to model some phenomena in financial mathematics, particularly in the modeling of options. For instance, the Black–Scholes equation is a parabolic partial differential equation that describes the price of the option over time and whose equation reads:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where V is the price of the option as a function of stock price S and time t, r is the risk-free interest rate and σ is the voatility of the stock. This option pricing model PDE can be transformed into the heat equation allowing relatively easy solutions.

Furthermore, the heat equation is also connected with Fokker–Planck equation, which describes the time evolution of the probability density function of the velocity of a particle under the influence of drag forces and random walks, as in Brownian motion.

The majority of financial models governed by PDEs work with a large number of independent variables, which makes them quite difficult to solve. This problem can be addressed by mapping the finance PDE to the heat equation generalized to more spatial dimensions.

A common and natural question that may arise is the following: Why is there a need to perform parameter inference and model discovery using quantum algorithms if these problems are already solved in a classical setting? Which advantages do PQCs offer with respect to PINNs? In fact, the implementation of a DQC for solving the problem stated in this project is justified by the fact that PQC models are a better option than PINNs when it comes to studying quantum mechanical systems and solving quantum problems. The PDE for excellence in the quantum world is the Time-Dependent Schrödinger equation 2, a linear partial differential equation that governs the wave function of a quantum-mechanical system.

Restricting to one dimension, the time-dependent Schrödinger equation is the governing equation for determining the wavefunction $\Psi(x, t)$ of a single non-relativistic particle with mass m.

Its most general form, including an arbitrary potential $U(x,t)$ is

$$i\hbar\frac{\partial\Psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\Psi(x,t)}{\partial x^2} + U(x,t)\Psi(x,t). \tag{11}$$

For a free particle $U = 0$, equation 11 reduces to

$$i\frac{\partial\Psi(x,t)}{\partial t} = -\frac{\hbar}{2m}\frac{\partial^2\Psi(x,t)}{\partial x^2}.$$

Rearranging terms, the previous equation can be written as

$$\frac{\partial\Psi(x,t)}{\partial t} = \frac{i\hbar}{2m}\frac{\partial^2\Psi(x,t)}{\partial x^2}.$$

This expression is formally similar to the particle diffusion equation 3, with $u(x,t) = \Psi(x,t)$ and $w = \frac{i\hbar}{2m}$. There exists an analogy between Schrödinger's equation for a free particle and the Heat equation, because the Schrödinger equation of quantum mechanics can be regarded as a heat equation in imaginary time. Hence, some quantum mechanics problems are also governed by a mathematical analog of the heat equation.

Nevertheless, this analogy between quantum mechanics and diffusion is purely a formal one. Physically, the evolution of the wave function satisfying Schrödinger's equation is not a diffusive process. But from a mathematical point of view, inferring the parameters of the Heat equation can be extended to the Schrödinger's equation, and this analogy can be very useful when modelling quantum systems.

8 Acknowledgements

First of all, I am immensely grateful to Dr. Jordi Tura Brugués for giving me the wonderful opportunity to join the Applied Quantum Algorithms (AQA) group at Lorentz Institute and to conduct a high level research project. Thanks for supervising my thesis and guiding me through the research process.

I would like to thank the other researchers of Lorentz Institute for hosting me during 7 months, accepting me as another member of the group and letting me work with them. Thanks for inviting me to participate in the weekly group activities (seminars, colloquia, talks and conferences), since it has enabled me to learn a lot about different science fields: not only quantum algorithms, but also condensed matter and astronomy.

Also, I am grateful for all the help received from all the AQA members, who have helped me countless times with my research and with whom I have maintained interesting and fruitful conversations during my stay. In particular, special mention to David Simon Dechant (who mentored and advised me with my work), Adrián Pérez Salinas (who helped me optimizing and speeding up my code) and Alberto Manzano Herrero (who helped me in my struggles with programming and machine learning). Thanks to David, Adrián and Alberto, with whom I have spent most of the time and from whom I have learned an immense amount of things.

Finally, special thanks to CFIS (Centre de Formació Interdisciplinària Superior) for making this amazing exchange opportunity possible and to Fundació Privada Cellex for providing financial support for my stay in such a prestigious institution as Leiden University.

9 Annexes

9.1 Introduction to Partial Differential Equations

A partial differential equation is an equation that involves n independent variables denoted by $\mathbf{x} = (x_1, x_2, x_3, ..., x_n \in \Omega \subseteq \mathbb{R}^n)$, a dependent function of these variables $u = u(x_1, ..., x_n) \in \mathbb{R}$, and the partial derivatives of the dependent function u with respect to the independent variables.

A solution (or a particular solution) to a partial differential equation is a function that solves the equation or, in other words, turns it into an identity when substituted into the equation. A solution is called general if it contains all particular solutions of the equation concerned.

Let's assume throughout that the solution u will be smooth enough so that partial derivatives commute. In this case, and for the sake of simplicity, the partial derivatives will be denoted as the following expressions:

$$u_1 = \frac{\partial u}{\partial x_1};\ u_2 = \frac{\partial u}{\partial x_2};\ u_{11} = \frac{\partial^2 u}{\partial x_1 \partial x_1} = \frac{\partial^2 u}{\partial x_1^2};\ u_{12} = \frac{\partial^2 u}{\partial x_1 \partial x_2}.$$

Then a partial differential equation is any equation of the form

$$F(u, x_1, ..., x_n, ..., u_1, ..., u_n, ..., u_{11}, u_{12}, u_{22}...) = 0$$

involving a finite number of derivatives.

Partial differential equations are used to formulate problems involving functions of several variables; such as the propagation of sound or heat, electrostatics, electrodynamics, fluid flow, and elasticity. In many physics problems, some of the independent variables typically correspond to space (denoted as x, y, z) and another to time (t).

9.1.1 Order of PDEs

The **order** of a partial differential equation is given by the order of the highest derivative involved. Some examples of PDEs are the following:

- First-order PDE: $u_t + u_x^3 + u^4 = 0$
- Second-order PDE: $u_{xx} + u_{yy} + u_{xy} = 0$
- Third-order PDE: $u_t - uu_{xxx} - \sin x = 0$

9.1.2 Linearity of PDEs

PDEs can be categorised into two: Quasi-linear and Non-Quasilinear. Quasi-linear PDEs are further categorised into two: Semi-linear, Non-semilinear. Semi-linear PDEs are further categorised into two: Linear and Nonlinear. One has the following picture:

$$\text{Linear PDE} \subsetneqq \text{Semi-linear PDE} \subsetneqq \text{Quasi-linear PDE} \subsetneqq \text{PDE}$$

A PDE is **linear** if it is of first degree in all of its variables and partial derivatives and all multiplicative factors are either constants or functions of the independent variables. A second-order

linear PDE with two independent variables has the form

$$A(x,y)u_{xx} + B(x,y)u_{xy} + C(x,y)u_{yy} + D(x,y)u_x + E(x,y)u_y + F(x,y)u = G.$$

For example: $u_{xx} + u_{yy} = x$.

A second-order **Semi-linear** PDE with two independent variables has the form:

$$A(x,y)u_{xx} + B(x,y)u_{xy} + C(x,y)u_{yy} = F(x,y,u,u_x,u_y).$$

A second-order **Quasi-linear** PDE with two independent variables has the form:

$$A(x,y,u)u_{xx} + B(x,y,u)u_{xy} + C(x,y,u)u_{yy} = F(x,y,u,u_x,u_y).$$

Any other form of PDE is said to be **Nonlinear**. The general form of a second-order nonlinear PDE with two independent variables is:

$$F(x,y,u,u_x,u_y,u_{xx},u_{xy},u_{yy}) = 0.$$

For instance: $u_x + u_y^2 = 0$.

The linearity classification helps identifying or guessing the properties of solutions of PDEs in that class. For instance, nonlinear differential equations model several phenomena but are notoriously difficult to solve. When it comes to quantum computing, there has been extensive previous work on efficient quantum algorithms for linear PDEs. However, since quantum mechanics is described by linear dynamics, the analogous progress for the nonlinear case is quite limited [47]. Hence, in the field of quantum algorithms, it is important to linearize nonlinear PDEs when possible.

9.1.3 Homogeneity of PDEs

Every linear PDE can be written in the form:

$$L(\mathbf{u}) = \mathbf{f}$$

where $\mathbf{u} \mapsto L(\mathbf{u})$ is a linear map and $\mathbf{f}$ is a function of independent variables only. The linear PDE is homogeneous if $\mathbf{f} = 0$. If $\mathbf{f} \neq 0$, the PDE is nonhomogeneous. For example:

- Homogeneous: $u_x + u_y + u_z + u_t = 0$
- Non-homogeneous: $u_x + u_y + u_z + u_t = x + y$

9.1.4 Classification of second order linear PDEs

Second order linear PDEs are widely studied because they model the vast majority of physical processes, and they can be classified into elliptic, parabolic or hyperbolic [48]. Generally, these PDEs are studied in two dimensions, with the most general case given by

$$A(x,y)u_{xx} + B(x,y)u_{xy} + C(x,y)u_{yy} + D(x,y)u_x + E(x,y)u_y + F(x,y)u = G.$$

The coefficients A, B, C are assumed not to vanish simultaneously, because in that case the second-order PDE degenerates to one of first order. Furthermore, as described in section 9.1.3,

if $G = 0$ the equation is homogeneous; otherwise, it is non-homogeneous. The function $u(x, y)$ and the coefficients are assumed to be twice continuously differentiable in some domain Ω.

The classification of second-order PDE depends on the form of the leading part of the equation consisting of the second order terms. So, for simplicity of notation, the lower order terms can be combined and the above equation is rewritten in the following form:

$$A(x, y)u_{xx} + B(x, y)u_{xy} + C(x, y)u_{yy} = \Phi(x, y, u, u_x, u_y).$$

Mathematically, the type of second-order PDE at a point (x_0, y_0) depends on the sign of the discriminant defined as:

$$\Delta(x_0, y_0) = B(x_0, y_0)^2 - 4A(x_0, y_0)C(x_0, y_0).$$

Then, the classification of the PDE is given by the following criteria:

1. Elliptic: $\Delta(x_0, y_0) < 0$
2. Parabolic: $\Delta(x_0, y_0) = 0$
3. Hyperbolic: $\Delta(x_0, y_0) > 0$

A given PDE may be of one type at a specific point and of another type at some other point. For example, the Tricomi equation $u_{xx} + xu_{yy} = 0$ has a discriminant equal to $\Delta = -4x$. Hence, it is hyperbolic in the left half-plane $x < 0$, parabolic for $x = 0$ and elliptic in the right half-plane $x > 0$.

A PDE is hyperbolic/parabolic/elliptic in a region Ω if the PDE is hyperbolic/parabolic/elliptic at each point of the domain Ω.

The terminology chosen to classify PDEs reflects the analogy between the form of the discriminant for PDEs and the form of the discriminant which classifies conic sections given by

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0.$$

The type of the curve represented by the above conic section depends on the sign of the discriminant, $\Delta = B^2 - 4AC$. If $\Delta > 0$, the curve is a hyperbola, if $\Delta = 0$ the curve is a parabola, and if $\Delta < 0$ the equation is an ellipse.

This classification can be generalized to n independent variables $x_1, ...x_n$ by taking into account the sign of the eigenvalues of the coefficient matrix. Then, the following classification holds:

1. Elliptic: all the eigenvalues have the same sign
2. Parabolic: one of the eigenvalues is zero
3. Hyperbolic: only one eigenvalue has a different sign from the rest

As an example, for an arbitrary 3-dimensional second-order linear PDE:

$$Au_{xx} + Bu_{yy} + Cu_{zz} + Du_{xy} + Eu_{xz} + Fu_{yz} = 0.$$

its coefficient matrix is given by

$$\begin{pmatrix} A & D/2 & E/2 \\ D/2 & B & F/2 \\ E/2 & F/2 & C \end{pmatrix}$$

The majority of physical phenomena are described by second order linear PDEs, with each type of equation modelling a different situation.

1. **Elliptic**: they are associated with a special state of a system, since they typically characterize steady-state systems (with no time derivative) and they represent equilibrium processes related to temperature, pressure, electrical potential, torsion and membrane displacement.

   The most known elliptic PDE is **Poisson**'s equation

   $$\Delta u = f.$$

   In 3-dimensional Cartesian coordinates, it takes the form:

   $$u_{xx} + u_{yy} + u_{zz} = f(x, y, z).$$

   Poisson's equation is a partial differential equation with broad utility in electrostatics, mechanical engineering and theoretical physics.

   When $f = 0$ it is called **Laplace** equation. The Laplace equation is often encountered in heat and mass transfer theory, fluid mechanics, elasticity, electrostatics, electrostatics, and other areas of mechanics mechanics and physics.

   $$\Delta u = 0.$$

   Another common elliptic PDE is **Helmholtz** equation, which models many problems related to steady state oscillations (mechanical, acoustical, thermal, electromagnetic).

   $$\Delta u + \lambda u = f.$$

   For $\lambda < 0$, this equation describes mass transfer processes with volume chemical reactions of first order.

2. **Parabolic**: they describe evolutionary phenomena that lead to a steady state described by an elliptic equation. They represent time-dependent diffusion processes with variation in both space and time (containing the first derivative in time). All changes are propagated across space at decreasing amplitudes and forward in time (backward propagation in time is not allowed).

   The most known parabolic PDE is the **Heat** equation.

   $$u_t = \alpha \Delta u.$$

   In 3-dimensional Cartesian coordinates, it takes the form:

   $$u_t = \alpha(u_{xx} + u_{yy} + u_{zz}).$$

   Other typical examples contain the Heat Conduction or Diffusion equation.

3. **Hyperbolic**: they model the transport and propagation of some physical quantity such as fluids or waves, with variation in both space and time (containing the second derivative in time). All changes are propagated forward in time (backward propagation in time is not allowed).

   The most known parabolic PDE is the **Wave** equation.

   $$u_{tt} = c^2 \Delta u.$$

   In 3-dimensional Cartesian coordinates, it takes the form:

   $$u_{tt} = c^2 (u_{xx} + u_{yy} + u_{zz}).$$

   Another common hyperbolic PDE is **Helmholtz** equation, which models many problems related to steady state oscillations (mechanical, acoustical, thermal, electromagnetic).

   $$\Delta u + \lambda u = f.$$

9.2 Derivation of the Parameter Shift Rule for gates with generators with two distinct eigenvalues

Consider a gate $\hat{U}_G(x) = e^{-iax\hat{G}}$ generated by a unitary and Hermitian operator $\hat{G}$ and with constant $a \in \mathbb{R}$. Then the generator G is also involutory ($\hat{G}^2 = \mathbb{1}$) and idempotent ($\hat{G}^2 = \hat{G}$).

Its derivative is given by

$$\partial_x \hat{U}_G = -ia\hat{G}e^{-iax\hat{G}}. \tag{12}$$

If $\hat{G}$ has just two distinct eigenvalues (which can be repeated) one can, without loss of generality, shift the eigenvalues to $\pm\lambda$, as the global phase is unobservable. Any single qubit gate is of this form.

Suppose that the Hermitian generator $\hat{G}$ of the unitary operator $\hat{U}_G(x) = e^{-iax\hat{G}}$ has at most two unique eigenvalues $\pm\lambda$ and $r = |\lambda|$. With Euler's identity, one can express the operator as

$$\hat{U}_G(x) = e^{-iax\hat{G}} = \sum_{k=0}^{\infty} \frac{(-iax)^k \hat{G}^k}{k!}$$

Separating into even and odd terms:

$$\hat{U}_G(x) = \sum_{k=0}^{\infty} \frac{(-iax)^{2k} \hat{G}^{2k}}{(2k)!} + \sum_{k=0}^{\infty} \frac{(-iax)^{2k+1} \hat{G}^{2k+1}}{(2k+1)!}.$$

The fact that $\hat{G}$ has the spectrum $\pm\lambda$ implies $\hat{G}^2 = \lambda^2\mathbb{1} = r^2\mathbb{1}$.

$$\hat{U}_G(x) = \mathbb{1}\sum_{k=0}^{\infty} \frac{(-1)^k (axr)^{2k}}{(2k)!} - ir^{-1}\hat{G}\sum_{k=0}^{\infty} \frac{(-1)^k (axr)^{2k+1}}{(2k+1)!}.$$

Therefore, the sine and cosine parts of the Taylor series of $\hat{U}_G$ take the form:

$$\hat{U}_G(x) = \mathbb{1}\cos(axr) - \frac{i}{r}\hat{G}\sin(axr).$$

As special case, one has

$$\hat{U}_G\left(\pm\frac{\pi}{4ar}\right) = \frac{1}{\sqrt{2}}\left(\mathbb{1} \mp \frac{i}{r}\hat{G}\right). \tag{13}$$

Let's take a general case of a quantum function

$$f(x) = \langle\psi|\, \hat{U}_G^\dagger(x)\hat{A}\hat{U}_G(x)\, |\psi\rangle\,.$$

Using the product rule, the partial derivative then looks

$$\partial_x f = \partial_x \langle\psi|\, \hat{U}_G^\dagger\hat{A}\hat{U}_G\, |\psi\rangle = \langle\psi|\, \hat{U}_G^\dagger\hat{A}(\partial_x\hat{U}_G)\, |\psi\rangle + \langle\psi|\, (\partial_x\hat{U}_G^\dagger)\hat{A}\hat{U}_G\, |\psi\rangle\,.$$

Using equation 12, one gets:

$$\partial_x f = \langle\psi|\, \hat{U}_G^\dagger\hat{A}(-ia\hat{G})\hat{U}_G\, |\psi\rangle + \langle\psi|\, (ia\hat{G})\hat{U}_G^\dagger\hat{A}\hat{U}_G\, |\psi\rangle\,.$$

Since $[\hat{A}, f(\hat{A})] = 0$, one has $[\hat{G}, \hat{U}_G^\dagger] = \hat{G}\hat{U}_G^\dagger - \hat{U}_G^\dagger\hat{G} = 0$ and so $\hat{G}\hat{U}_G^\dagger = \hat{U}_G^\dagger\hat{G}$. Taking $|\phi\rangle = \hat{U}_G|\psi\rangle$ and rearranging terms:

$$\partial_x f = ar\left[\langle\phi|\hat{A}\frac{-i}{r}\hat{G}|\phi\rangle + \langle\phi|\frac{i}{r}\hat{G}\hat{A}|\phi\rangle\right].$$

For any two operators $\hat{B}$ and $\hat{C}$, the following relation holds [37]:

$$\langle\psi|\hat{B}^\dagger\hat{A}\hat{C}|\psi\rangle + \langle\psi|\hat{C}^\dagger\hat{A}\hat{B}|\psi\rangle = \frac{1}{2}\left[\langle\psi|(\hat{B}+\hat{C})^\dagger\hat{A}(\hat{B}+\hat{C})|\psi\rangle - \langle\psi|(\hat{B}-\hat{C})^\dagger\hat{A}(\hat{B}-\hat{C})|\psi\rangle\right]. \tag{14}$$

Using equation 14 for $\hat{B} = \mathbb{1}$ and $\hat{C} = \frac{-ia}{r}\hat{G}$ one can write

$$\partial_x f = a\frac{r}{2}\left[\langle\phi|(\mathbb{1} - \frac{i}{r}\hat{G})^\dagger\hat{A}(\mathbb{1} - \frac{i}{r}\hat{G})|\phi\rangle - \langle\phi|(\mathbb{1} + \frac{i}{r}\hat{G})^\dagger\hat{A}(\mathbb{1} + \frac{i}{r}\hat{G})|\phi\rangle\right].$$

One can recognize from 13 that these unitaries represent instances of the initial gate.

$$\partial_x f = a\frac{r}{2}\left[\langle\phi|\sqrt{2}\hat{U}_G\left(\frac{\pi}{4ar}\right)^\dagger\hat{A}\sqrt{2}\hat{U}_G\left(\frac{\pi}{4ar}\right)|\phi\rangle - \langle\phi|\sqrt{2}\hat{U}_G\left(-\frac{\pi}{4ar}\right)^\dagger\hat{A}\sqrt{2}\hat{U}_G\left(-\frac{\pi}{4ar}\right)|\phi\rangle\right].$$

Undoing the change of state $|\phi\rangle = \hat{U}_G|\psi\rangle$, one gets:

$$\partial_x f = ar\left[\langle\psi|\hat{U}_G^\dagger\hat{U}_G\left(\frac{\pi}{4ar}\right)^\dagger\hat{A}\hat{U}_G\left(\frac{\pi}{4ar}\right)\hat{U}_G|\psi\rangle - \langle\psi|\hat{U}_G^\dagger\hat{U}_G\left(-\frac{\pi}{4ar}\right)^\dagger\hat{A}\hat{U}_G\left(-\frac{\pi}{4ar}\right)\hat{U}_G|\psi\rangle\right].$$

Since for unitarily generated one-parameter gates $\hat{G}(a)\hat{G}(b) = \hat{G}(a+b)$, this is equivalent to shifting the gate parameter a shift $s = \frac{\pi}{4ar}$.

$$\partial_x f = ar\left[\langle\psi|\hat{U}_G^\dagger(x+s)\hat{A}\hat{U}_G(x+s)|\psi\rangle - \langle\psi|\hat{U}_G^\dagger(x-s)\hat{A}\hat{U}_G(x-s)|\psi\rangle\right].$$

Then, one gets the "parameter shift rule"

$$\partial_x f = ar\left[f(x+s) - f(x-s)\right].$$

The parameter shift rule applies to a number of special cases. The set of Pauli matrices $\mathbb{1}, \hat{X}, \hat{Y}, \hat{Z}$ are generators with two distinct eigenvalues, namely $\pm\lambda = \pm 1$. For instance, the 1-qubit Pauli rotation gates are all parameter-shift differentiable with $a = \frac{1}{2}$.

$$\hat{R}_X(\alpha) = e^{-i\frac{1}{2}\alpha X};\ X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\hat{R}_Y(\alpha) = e^{-i\frac{1}{2}\alpha Y};\ Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$\hat{R}_Z(\alpha) = e^{-i\frac{1}{2}\alpha Z};\ Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The shift constant for all of them is $s = \frac{\pi}{2}$. Hence, the parameter shift rule becomes

$$\partial_x f = \frac{1}{2}\left[f(x + \frac{\pi}{2}) - f(x - \frac{\pi}{2})\right].$$

9.3 *Matlab* code

```
%% Generating data u(x,t) by solving a PDE

% Heat equation: Fw= w*uxx - ut = 0
x_points= 20;
t_points= 10;
x= linspace(0,0.9,x_points);
t= linspace(0,0.1,t_points);
m=0;

u= pdepe(m, @heatpde, @heatic, @heatbc, x, t);
save('u_heat.mat', 'u')
save('x_heat.mat', 'x')
save('t_heat.mat', 't')

function [c,f,s] = heatpde(x,t,u,dudx)
w=1;

c= 1;
f= w*dudx;
s= 0;
end

function u0 = heatic(x)
u0=sin(pi/0.9*x);
end

function [pl, ql, pr, qr] = heatbc(xl, ul, xr, ur, t)
pl= ul;
ql=0;
pr= ur;
qr=0;
end
```

9.4 *Python* code

```
#Do the necessary imports

import numpy as np
import scipy.io
import sys
from scipy.optimize import minimize, minimize_scalar, Bounds
import pickle

import matplotlib.pyplot as plt
from time import time

from sklearn.metrics import mean_squared_error, mean_absolute_error


class PQC():
    Z= np.matrix([[1,0], [0,-1]])
    CNOT= np.matrix([[1,0,0,0], [0,1,0,0], [0,0,0,1], [0,0,1,0]])

    def __init__(self, n_qubits, n_layers, error, weights, params=None, seed=42):
        self.n_qubits= n_qubits
        self.n_layers= n_layers
        self.error= error
        self.weights= weights
        self.theta_params = self.n_qubits * 3 * self.n_layers

        self.Hamiltonian_operator= np.kron(np.eye(2**(self.n_qubits-1), 2**(self.n_qubits-1)), self.Z)

        self.L_w=[]
        self.ws=[]

        np.random.seed(seed)
        #Discover_w
        #self.w0 = np.random.uniform(-10, 10, size=1)[0]

        #PQC
        self.w0=1

        if params == None:
            self.params = np.random.uniform(0, 2*np.pi, size=self.theta_params)
        else:
            self.params= params

        self.set_params(self.params)

    def load_grid (self, u_file= 'u_heat.mat', x_file='x_heat.mat', t_file='t_heat.mat'):
        self.u_file= u_file
        self.x_file= x_file
        self.t_file= t_file
        _u = scipy.io.loadmat(self.u_file) # u loads as a dictionary
        _x = scipy.io.loadmat(self.x_file) # u loads as a dictionary
        _t = scipy.io.loadmat(self.t_file) # u loads as a dictionary
        self.u_real= np.transpose(_u['u']) #x rows and t columns
        self.x_real= np.transpose(_x['x'])
        self.t_real= _t['t']
        self.x_points= self.x_real.size #Number of space data points
        self.t_points= self.t_real.size #Number of time data points
```

```
57
58          Xl= np.full((self.t_points,2), self.x_real[0][0])
59          for i in range(self.t_points):
60              Xl[i][0] = self.t_real[0][i]
61
62          Xr= np.full((self.t_points,2), self.x_real[-1][0])
63          for i in range(self.t_points):
64              Xr[i][0] = self.t_real[0][i]
65
66          T0= np.full((self.x_points-2,2), self.t_real[0][0])
67          for i in range(self.x_points-2):
68              T0[i][1] = self.x_real[i+1][0]
69
70          self.BC_Points= np.concatenate([Xl, Xr, T0])
71          Ul= self.u_real[0]
72          Ur= self.u_real[-1]
73          U0= np.transpose(self.u_real)[0][1:-1]
74          self.BC_Data= np.concatenate([Ul, Ur, U0])
75
76          self.T=np.full((self.x_points,self.t_points), self.t_real) # x_points x
    t_points
77          self.X= np.full((self.x_points, self.t_points), self.x_real)
78
79          self.Points = np.transpose((self.T.flatten(), self.X.flatten())) #Cada
    element: [t x]
80          Y = self.u_real.reshape((self.u_real.size, 1))
81
82
83          #Add noise
84          noise_level = 0
85
86          #Gaussian noise
87          #std = noise_level * np.std(Y) # for %5 Gaussian noise
88          #mu=0
89          #noise = np.random.normal(mu, std, size = Y.shape)
90          #y_noisy = Y + noise
91
92          #Normal distribution
93          y_noisy = Y + noise_level * np.std(Y) * np.random.randn(Y[:,0].size, 1)
94
95          self.Y_noisy= np.transpose(y_noisy)
96
97          self.N= self.Y_noisy.size #Number of total data points
98          self.M= self.BC_Data.size #Number of total bc data points
99
100         self.initials = self.initialize_grid(self.Points)
101         self.BC_initials= self.initialize_grid(self.BC_Points)
102
103     def save_params(self, filename): #save params in a file
104         with open(filename, 'wb') as outp:  # Overwrites any existing file.
105             pickle.dump(self.params, outp)
106
107     def load_params(self, filename): #load params from a file
108         with open(filename, 'rb') as inp:
109             #self.params= pickle.load(inp)
110             self.set_params(self, pickle.load(inp))
111
112
113     def set_params(self, params):
114         self.params = params
```

```
        self.set_U_theta()

    #Define the nonlinearity function
    def phi(self, x_i, j): #Chebyshev
        return 2*j*np.arccos(x_i)

    def dphi1(self, x):
        return -1/np.sqrt(1-x**2)

    def dphi2(self, x):
        return -x / ((1-x**2)**(3/2))

    def Rx(self, alpha):
        return np.matrix([[np.cos(alpha/2), -1j*np.sin(alpha/2)], [-1j*np.sin(
    alpha/2), np.cos(alpha/2)]])

    def Ry(self, alpha):
        return np.matrix([[np.cos(alpha/2), -np.sin(alpha/2)], [np.sin(alpha/2),
     np.cos(alpha/2)]])

    def Rz(self, alpha):
        return np.matrix([[np.exp(-1j*alpha/2), 0], [0, np.exp(1j*alpha/2)]])

    #Variational ansatz --> Hardware efficient ansatz (HEA)
    def set_U_theta(self): #variational_circuit
        U_theta= np.eye(2**self.n_qubits)
        for d in range(self.n_layers):
            R1=1
            R2=1
            R3=1
            for j in range(self.n_qubits):
                R1= np.kron(R1, self.Rz(self.params[j + 3*self.n_qubits*d]))
                R2= np.kron(R2, self.Rx(self.params[j+ self.n_qubits*(1+ 3*d)]))
                R3= np.kron(R3, self.Rz(self.params[j+ self.n_qubits*(2+ 3*d)]))

            C1=1
            C2= np.eye(2)
            if self.n_qubits%2==0:
                for j in range(int(self.n_qubits/2)-1):
                    C1= np.kron(C1, self.CNOT)
                    C2= np.kron(C2, self.CNOT)
                C1= np.kron(C1, self.CNOT)
                C2= np.kron(C2, np.eye(2))
            else:
                for j in range(int((self.n_qubits-1)/2)):
                    C1= np.kron(C1, self.CNOT)
                    C2= np.kron(C2, self.CNOT)
                C1= np.kron(C1, np.eye(2))

            M= C2 @ C1 @ R3 @ R2 @ R1
            U_theta= M @ U_theta

        self.U_theta = U_theta

    # Encoding layer that uses x_i as angles to encode data.
    def single_encoding(self, qubits, x, J, sign): # J(vector[j1, j2]), sign(
    vector with the signs of deltas)
        aux=1
        for i in range(qubits):
            deltas=0
```

```
            for j in range(len(J)):
                deltas += (i+1 == J[j])*sign[j]
            shift= deltas%4
            s= self.phi(x, i+1) + shift * np.pi/2
            aux= np.kron(aux, self.Ry(s))
        return aux

    #Matrix U_phi
    def U_phi(self, Xi, J, sign, dvar):
        ti, xi = Xi
        assert self.n_qubits % 2 == 0

        if dvar==0:
            U1 = np.kron(self.single_encoding(self.n_qubits // 2, ti, [], []), self.single_encoding(self.n_qubits // 2, xi, [], []))
        elif dvar=='t':
            U1 = np.kron(self.single_encoding(self.n_qubits // 2, ti, J, sign), self.single_encoding(self.n_qubits // 2, xi, [], []))
        elif dvar=='x':
            U1 = np.kron(self.single_encoding(self.n_qubits // 2, ti, [], []), self.single_encoding(self.n_qubits // 2, xi, J, sign))

        #U1 = np.kron(self.single_encoding(self.n_qubits // len(Xi), Xi[0], [], []), self.single_encoding(self.n_qubits // len(Xi), Xi[1], [], []))
        #for i in range(len(Xi)-2):
        #    U1 = np.kron(U1, self.single_encoding(self.n_qubits // len(Xi), Xi[i+2], [], []))
        return U1

    #State |psi> after U_phi
    def psi_phi(self, Xi, J, sign, dvar):
        return self.U_phi(Xi, J, sign, dvar)[:,0]

    def create_initials(self, Xi):
        Psi0 = self.psi_phi(Xi, [], [], 0)

        Psi1_1 = []
        Psi1_2 = []
        for j in range(self.n_qubits):
            J= [j+1]
            Psi1_1.append([self.psi_phi(Xi, J, [1], 't'), self.psi_phi(Xi, J, [1], 'x')])
            Psi1_2.append([self.psi_phi(Xi, J, [-1], 't'), self.psi_phi(Xi, J, [-1], 'x')])


        Psi2_1 = []
        Psi2_2 = []
        Psi2_3 = []
        Psi2_4 = []
        for j2 in range(self.n_qubits):
            for j1 in range(self.n_qubits):
                J= [j1+1, j2+1]
                Psi2_1.append([self.psi_phi(Xi, J, [1,1], 't'),self.psi_phi(Xi, J, [1,1], 'x')])
                Psi2_2.append([self.psi_phi(Xi, J, [-1,1], 't'),self.psi_phi(Xi, J, [-1,1], 'x')])
                Psi2_3.append([self.psi_phi(Xi, J, [1,-1], 't'),self.psi_phi(Xi, J, [1,-1], 'x')])
                Psi2_4.append([self.psi_phi(Xi, J, [-1,-1], 't'),self.psi_phi(Xi
```

```
    , J, [-1,-1], 'x')])

        return Psi0, Psi1_1, Psi1_2, Psi2_1, Psi2_2, Psi2_3, Psi2_4

    def initialize_grid(self, grid):
        initials = [[]] * len(grid)
        for i, Xi in enumerate(grid):
            initials[i] = list(self.create_initials(Xi))

        return initials

    #Calculate expectation value
    def expectation(self, operator, state):
        exp= state.T.conj() @ operator @ state
        return exp.real
    #|state> = U_theta * U_phi |0n>
    # operator= C

    #Measure
    def output(self, in_state):
        out_state= self.U_theta @ in_state
        return self.expectation(self.Hamiltonian_operator, out_state)[0,0]

    #Parameter shift rule
    def ps1(self, elem, dvar):
        count1=0
        if dvar == 't': dvar = 0
        elif dvar == 'x': dvar = 1

        Psi0, Psi1_1, Psi1_2, Psi2_1, Psi2_2, Psi2_3, Psi2_4 = self.initials[
    elem]
        k=0
        for j in range(self.n_qubits):
            out1= self.output(Psi1_1[k][dvar])
            out2= self.output(Psi1_2[k][dvar])
            count1 += (j+1)*(out1-out2)
            k+=1
        return count1

    #Parameter shift rule applied twice
    def ps2 (self, elem, dvar):
        count2=0
        if dvar == 't': dvar = 0
        elif dvar == 'x': dvar = 1

        Psi0, Psi1_1, Psi1_2, Psi2_1, Psi2_2, Psi2_3, Psi2_4 = self.initials[
    elem]

        k = 0
        for j2 in range(self.n_qubits):
            for j1 in range(self.n_qubits):
                out1= self.output(Psi2_1[k][dvar])
                out2= self.output(Psi2_2[k][dvar])
                out3= self.output(Psi2_3[k][dvar])
                out4= self.output(Psi2_4[k][dvar])
                count2 += (j1+1)*(j2+1)*(out1 - out2 - out3 + out4)
                k += 1
        return count2

    def ui(self, elem):
```

```
279         Psi0, Psi1_1, Psi1_2, Psi2_1, Psi2_2, Psi2_3, Psi2_4 = self.initials[
    elem]
280         return self.output(Psi0)
281 
282     def uBC(self, elem):
283         BC_Psi0, BC_Psi1_1, BC_Psi1_2, BC_Psi2_1, BC_Psi2_2, BC_Psi2_3,
    BC_Psi2_4 = self.BC_initials[elem]
284         return self.output(BC_Psi0)
285 
286     #First derivative
287     def u_i(self, elem, dvar):
288         if dvar== 't': return 1/2*self.ps1(elem, dvar)*self.dphi1(self.Points[
    elem][0])
289         elif dvar== 'x': return 1/2*self.ps1(elem, dvar)*self.dphi1(self.Points[
    elem][1])
290 
291     #Second derivative
292     def u_ii(self,elem, dvar):
293         if dvar== 't': return self.ps2(elem,dvar)*1/4*(self.dphi1(self.Points[
    elem][0]))**2 + 1/2*self.ps1(elem,dvar)*self.dphi2(self.Points[elem][0])
294         elif dvar== 'x': return self.ps2(elem,dvar)*1/4*(self.dphi1(self.Points[
    elem][1]))**2 + 1/2*self.ps1(elem,dvar)*self.dphi2(self.Points[elem][1])
295 
296 
297     def set_lambda(self, weights):
298         if weights==0:
299             lambda_f = (1/self.N)/(1/self.N + 2/self.t_points + 1/(len(self.
    BC_Data)-2*self.t_points))
300             lambda_bl= (1/self.t_points)/(1/self.N + 2/self.t_points + 1/(len(
    self.BC_Data)-2*self.t_points))
301             lambda_br= (1/self.t_points)/(1/self.N + 2/self.t_points + 1/(len(
    self.BC_Data)-2*self.t_points))
302             lambda_b0 = (1/(len(self.BC_Data)-2*self.t_points))/(1/self.N + 2/
    self.t_points + 1/(len(self.BC_Data)-2*self.t_points))
303         elif weights==1:
304             lambda_f = self.N/(self.N + 2*self.t_points + (len(self.BC_Data)-2*
    self.t_points))
305             lambda_bl= self.t_points/(self.N + 2*self.t_points + (len(self.
    BC_Data)-2*self.t_points))
306             lambda_br= self.t_points/(self.N + 2*self.t_points + (len(self.
    BC_Data)-2*self.t_points))
307             lambda_b0 = (len(self.BC_Data)-2*self.t_points)/(self.N + 2*self.
    t_points + (len(self.BC_Data)-2*self.t_points))
308 
309         return lambda_f, lambda_bl, lambda_br, lambda_b0
310 
311     def norm_inf(self, x,y):
312         return np.max(np.abs(x-y))**2
313 
314     def __call__(self,i):
315         u = self.ui(i)
316         ut = self.u_i(i, 't')
317         ux = self.u_i(i, 'x')
318         uxx = self.u_ii(i, 'x')
319 
320         return u, ut, ux, uxx
321 
322     def predict(self):
323         u= np.zeros(self.N)
324         ut= np.zeros(self.N)
```

```
325         ux= np.zeros(self.N)
326         uxx= np.zeros(self.N)
327         for i, Xi in enumerate(self.Points):
328             u[i], ut[i], ux[i], uxx[i] = self(i)
329         return u, ut, ux, uxx
330 
331     def predict_BC(self):
332         u_bc=np.zeros(self.M)
333         for i, Xi in enumerate(self.BC_Points):
334             u_bc[i] = self.uBC(i)
335         return u_bc
336 
337 
338     def cost_theta (self, u, ut, ux, uxx, u_bc):
339 
340         lambda_f, lambda_bl, lambda_br, lambda_b0 = self.set_lambda(self.weights
    )
341 
342         if self.error=='MSE': #Mean Squared Error
343             norm = mean_squared_error
344         elif self.error=='MAE': #Mean Absolute Error
345             norm = mean_absolute_error
346         elif self.error=='MAX': #Max absolute
347             norm = self.norm_inf
348 
349         #Discover_w
350         #Fw= self.w*uxx-ut
351         #Fw= w1*u + w2*ux + w3*uxx
352 
353         #PQC
354         Fw= self.w0*uxx-ut
355 
356         function_loss= norm(Fw, np.zeros_like(Fw))
357         L_diff= lambda_f*function_loss
358 
359         L_Xl= lambda_bl*norm(u_bc[0:self.t_points], self.BC_Data[0:self.t_points
    ])
360         L_Xr= lambda_br*norm(u_bc[self.t_points:2*self.t_points], self.BC_Data[
    self.t_points:2*self.t_points])
361         L_T0= lambda_b0*norm(u_bc[2*self.t_points:len(u_bc)], self.BC_Data[2*
    self.t_points:len(self.BC_Data)])
362 
363         cost= L_diff + L_Xl + L_Xr + L_T0
364         self.L_cost= np.array([cost, L_diff, L_Xl, L_Xr, L_T0])
365 
366         return cost
367 
368     def cost_w (self, u):
369         data_loss= mean_squared_error(u, self.Y_noisy[0])
370         return data_loss
371 
372 
373     def loss_theta(self, theta):
374         self.set_params(theta)
375         u, ut, ux, uxx = self.predict()
376         u_bc = self.predict_BC()
377 
378         return self.cost_theta (u, ut, ux, uxx, u_bc)
379 
380     def loss_w (self, w):
```

```
381         self.w= w
382         optimal_theta = self.fit_theta(inplace=True)
383         self.set_params(optimal_theta)
384         u, ut, ux, uxx = self.predict()
385         self.L_w.append(self.cost_w(u))
386         self.ws.append(self.w)
387 
388         return self.cost_w(u)
389 
390     def callback_theta(self, theta):
391         print('Loss function:', self.loss_theta(theta))
392 
393 
394     def fit_theta(self, inplace = True):
395         old_parameters = np.copy(self.params)
396         optimal_theta = minimize(self.loss_theta, self.params, method="L-BFGS-B"
    , bounds= Bounds(lb=0, ub=2*np.pi), callback= self.callback_theta(self.
    params))
397 
398         if inplace:
399             self.params = np.copy(optimal_theta.x)
400         else:
401             self.params = np.copy(old_parameters)
402 
403         return optimal_theta.x
404 
405     def fit_w(self):
406         #optimal_w = minimize(self.loss_w, self.w0, method="L-BFGS-B")
407         optimal_w = minimize_scalar(self.loss_w)
408         return optimal_w.x
```

Bibliography

[1] Marcello Benedetti, Erika Lloyd, Stefan Sack, and Mattia Fiorentini. Parameterized quantum circuits as machine learning models. *Quantum Science and Technology*, 4, November 2019.

[2] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, Xiao Yuan, L. Cincio, and P. J. Coles. Variational quantum algorithms. *Nature Review Physics*, 3(625), October 2021.

[3] Oleksandr Kyriienko, Paine Annie E., and Vincent E. Elfving. Solving nonlinear differential equations with differentiable quantum circuits. *Physical Review A*, 103:052416, May 2021.

[4] Ronald de Wolf. Quantum computing: Lecture notes, January 2022.

[5] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, New York, USA, 2011.

[6] P.W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41:303–332, 1999.

[7] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, July 2018.

[8] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2016.

[9] F. et al. Benatti. *Quantum Information, Computation and Cryptography: An Introductory Survey of Theory, Technology and Experiments*. 2010.

[10] John Preskill. Fault-tolerant quantum computation. *arXiv:9712048* [*quant-ph*], 1997.

[11] D. Gottesman. An introduction to quantum error correction and fault-tolerant quantum computation. *Proceedings of Symposia in Applied Mathematics*, 68, 2010.

[12] A. Harrow, A. Hassidim, and S. Lloyd. Quantum algorithm for solving linear systems of equations. *Physical Review Letters*, 15, 2009.

[13] Ashley Montanaro. Quantum algorithms for pdes. School of Mathematics, University of Bristol, November 2019.

[14] K. Sankara Rao. *Introduction to Partial Differential Equations*. PHI Learning Private Limited, New Delhi, 2011.

[15] D.P. García, J. Cruz-Benito, and F.J. García-Peñalvo. Systematic literature review: Quantum machine learning and its applications. *arXiv:2201.04093* [*quant-ph*], January 2022.

[16] Agnès Tourin. An introduction to finite difference methods for pdes in finance, March 2011.

[17] Gert-Jan Both. *Model Discovery of Partial Differential Equations*. PhD thesis, Université Paris PSL, December 2021.

[18] G.E. Karniadakis, I.G. Kevrekidis, and L. et al. Lu. Physics-informed machine learning. *Nature Review Physics*, 3:422–440, May 2021.

[19] I. E. Lagaris, A. Likas, and D. I. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000, September 1998.

[20] Andrew Childs, Jin-Peng Liu, and Aaron Ostrander. High-precision quantum algorithms for partial differential equations. *arXiv:2002.07868* [*quant-ph*], 2021.

[21] Seth Lloyd. Universal quantum simulators. *Science*, pages 1073–1078, 1996.

[22] Scott Aaronson. Read the fine print. *Nature Physics*, 11, 2015.

[23] Arthur Pesah. Quantum algorithms for solving partial differential equations. University College London, March 2020.

[24] Yudong Cao. Quantum algorithm and circuit design solving the poisson equation. *New Journal of Physics*, 15, 2013.

[25] Noah Linden, Ashley Montanaro, and Changpeng Shao. Quantum vs. classical algorithms for solving the heat equation. *arXiv:2004.06516* [*quant-ph*], 2020.

[26] Pedro C.S. Costa, Stephen Jordan, and Aaron Ostrander. Quantum algorithm for simulating the wave equation. *arXiv:1711.05394* [*quant-ph*], 2019.

[27] F. Fontanela, A. Jacquier, and M. Oumgari. A quantum algorithm for linear pdes arising in finance. *arXiv:1912.02753* [*q-fin.CP*], 2021.

[28] Dong An, Noah Linden, Jin-Peng Liu, Ashley Montanaro, Changpeng Shao, and Jiasu Wang. Quantum-accelerated multilevel monte carlo methods for stochastic differential equations in mathematical finance. *arXiv:2012.06283* [*quant-ph*], 2021.

[29] F. Oz, R. Vuppala, K. Kara, and F. Gaitan. Solving burgers' equation with quantum computing. *Quantum Information Processing*, 30, 2021.

[30] Robert König. Quantum advantage with shallow circuits. *Science*, 362(6412):308–311, October 2018.

[31] Maziar Raissi, Paris Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, February 2019.

[32] Lluis Alberto Bonals Muntada. Transferència de calor. https://upcommons.upc.edu/bitstream/handle/2117/90176, March 2020. Accessed: 2022-08-16.

[33] X. Xun, J. Cao, B. Mallick, RJ. Carroll, and A. Maity. Parameter estimation of partial differential equation models. *J Am Stat Assoc*, 108, April 2013.

[34] Andrea Mari, Thomas R. Bromley, and Nathan Killoran. Estimating the gradient and higher-order derivatives on quantum hardware. *Physical Review A*, 103:012405, February

2021.

[35] Oleksandr Kyriienko and Vincent E. Elfving. Generalized quantum circuit differentiation rules. *arXiv:2108.01218* [*quant-ph*], October 2021.

[36] Gavin Crooks. Gradients of parameterized quantum gates using the parameter-shift rule and gate decomposition. *arXiv:1905.13311* [*quant-ph*], May 2019.

[37] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. Evaluating analytic gradients on quantum hardware. *Physical Review A*, 99, March 2019.

[38] K. Mitarai, M. Negoro, M. Kitagawa, and K. Fujii. Quantum circuit learning. *Physical Review A*, 98:032309, September 2018.

[39] Nicklas Heim, Atiyo Ghosh, Oleksandr Kyriienko, and Vincent E. Elfving. Quantum model-discovery. *arXiv:2111.06376* [*quant-ph*], November 2021.

[40] Oleksandr Kyriienko, Annie E. Paine, and Vincent E. Elfving. Quantum quantile mechanics: Solving stochastic differential equations for generating time-series. *arXiv:2108.03190* [*quant-ph*], October 2021.

[41] Gaël Varoquaux. Scipy lectures. mathematical optimization: finding minima of functions. https://scipy-lectures.org/advanced/mathematical_optimization/. Accessed: 2022-08-10.

[42] The SciPy community. Scipy documents. optimization and root finding. https://docs.scipy.org/doc/scipy/reference/optimize.minimize-lbfgsb.html#optimize-minimize-lbfgsb. Accessed: 2022-08-10.

[43] The SciPy community. Scipy documents. minimize scalar. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize_scalar.html#scipy.optimize.minimize_scalar. Accessed: 2022-08-10.

[44] MathWorks. Solving partial differential equations. https://www.mathworks.com/help/matlab/math/partial-differential-equations.html. Accessed: 2022-08-10.

[45] R. Sweke, F. Wilde, J. Meyer, M. Schuld, P. K. Fährmann, B. Meynard-Piganeau, and J. Eisert. Stochastic gradient descent for hybrid quantum-classical optimization. *Quantum*, 4, January 2021.

[46] Leonardo Banchi and Gavin E. Crooks. Measuring analytic gradients of general quantum evolution with the stochastic parameter shift rule. *Quantum*, 5:386, January 2021.

[47] Jin-Peng Liu, Herman Kolden, Hari K. Krovi, Nuno F. Loureiro, Konstantina Trivissa, and Andrew M. Childs. Efficient quantum algorithm for dissipative nonlinear differential equations. *PNAS*, 118(35), August 2021.

[48] A. Salih. Classification of partial differential equations and canonical forms. Indian Institute of Space Science and Technology, December 2014.