Quantum Computing research group | Amsterdam University of Applied Sciences

# Pasqal

Justin Hoole, Salo van Winsen
Amsterdam University of Applied Science
Amsterdam, The Netherlands
{justin.hoole, salo.van.winsen}@hva.nl

June 29, 2022

**Summary**

Differentiable Quantum Circuit or DQC is a method of combining classical neural networks with the processing powers of the quantum computer. DQC is build from several components such as the data encoder, the variational quantum circuit (ansatz) and the classical neural optimizer. This paper discusses all of these topics in differing amounts of detail. With the primary focus being on the data encoding part of the DQC while creating enough knowledge about the other components to understand the system as a whole. The concepts for data encoding and their implementations are described for both the Python and Julia programming languages, where Python is used in combination with the Pennylane and torch libraries and Julia with the YAO package.

# Contents

# 1 Introduction

This paper is split up into several segments. Starting with a brief introduction to the company Pasqal as well as the goals of this project in section 2. Followed up by a short recap of the work done by the previous group in section 3.

Then multiple of the technologies used during this project are described and explained in section 4. Within this section Subsections 4.2 and 4.1 describe the Physics informed neural networks (PINNs) and their use within Differentiable Quantum Circuits (DQC). Subsection 4.3 explains the usage of and the possibility's within the Julia programming language. Subsection 4.4 gives a look at the possible feature maps that can be used. Followed up by the discussion about the different goals of this project in section 5. This is done through the lens of what is accomplished and what still has to be achieved. Finally, giving a conclusion about the topics discussed.

# 2 Project

Project Pasqal is one of the primary projects in the minor Applied Quantum Computing at the Amsterdam University of Applied Sciences (HvA). The project is done in collaboration the with Pasqal, a research company focused on the different aspects of quantum computing. Originally only focusing on the hardware side of Quantum Computing, Pasqal merged with Qu&Co in 2021 for their software expertise.

Now Pasqal will deliver a 1000-qubit quantum solution in 2023, on pace with the announced road maps of the most advanced quantum platforms.[1]

The project started with focusing primarily on the concepts of the PINN and the DQC as described in sections 4.2 and 4.1 respectively. During sprints 2 and 3 the goal of the project was adjusted. Focussing more on the implementation of feature maps and studying their expressivity. With the goal being further described in the upcoming section 2.1.

The project will be supported by Bernardo Villalba Frias of the HvA and Andrea Gentile of Pasqal.

## 2.1 Goals

The Pasqal project has goals to work towards. These goals are divided into primary goals and secondary goals.

As stated in Section 1 the first sprints where focussed on understanding the terms DQC and PINN's. This resulted in a lot of time and effort went in to reading papers and trying to understand the work the previous group had done before. At the end of sprint 2 a meeting has taken place with Andrea from Pasqal. During the meeting the project goals where adjusted. With the new goals stated below.

### 2.1.1 Primary goals

Understand and reproduce the feature mapping in a DQC architecture. By understanding the main difference of feature maps used in DQC against amplitude encoding, reproduce at least a feature map (FM)( – e.g. a product FM - with an architecture of choice (e.g. Python + Qiskit)).

### 2.1.2 Secondary goals

Expand the architecture implementations, investigating either: Alternative FMs( - such as Chebyshev, Fourier, Chebyshev-tower, . . . ), Alternative SW architectures( such as Python+qutip / Julia+Yao / Python+Pennylane. . . and study computational performances against "Implementation 0")

Study the expressivity of the FM of choice, by simple metrics such as overlaps of fully evolved quantum states mapping different independent variable values, for some exemplary target functions (which would represent the solutions of the ODE/PDE in a full DQC setting).

# 3 Previous group

The previous team worked on implementing a DQC with the Python/Pennylane package. A detailed step by step creation of their DQC structure can be found in their paper [2] at the repository [1].

The repository also includes a Jupyter notebook which contains the original code on which the Python part of this project is based upon. The code in this notebook starts with the declaration of a simple nonlinear differential equation. This equation was also used by the original team at Qu&Co. A dataset was provided with the results obtained by this original team. These results are used to compare the final results of the project at the end of their minor.

At this point the start of a quantum circuit with neural network is present, however missing the following things:

1. `Parameters for Adam`
2. `total magnetization`
3. `Boundary handling`
4. `parameter shift rule`

With an incomplete implementation of the Chebyshev Tower feature and ansatz. However, much of the work done during this project is based upon the work of this previous group. Furthermore, the paper contains a great deal of information about ansatz which is very useful in their understanding.

# 4 Technology

In this section a description of the different technologies used in this project will be given. Starting with explanations about some bigger concepts used within the project such as the DQC and PINN. Followed up by a description of one of the programming languages used, Julia, as it is relatively unknown. Finally, a section further describing some methods implemented to make it all work.

## 4.1 DQC

DQC or Differentiable Quantum Circuits are a form of quantum circuit with the goal of making quantum algorithms differentiable, and thus trainable. This allows for the integration of quantum circuits into a classical machine learning background. DQC can be generalized into two different methods all based on the same concept of variational quantum ansatz or simply ansatz. Hardware efficient and alternating blocks [2].

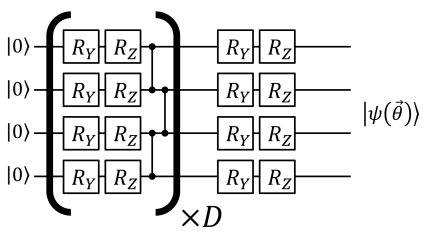
The quantum circuit of the hardware efficient ansatz consists of a sequence of single qubit rotation gates and two qubit gates aimed at entanglement. The number of repetitions of the single qubit rotations and two qubit gates is called the depth, often depicted as $D$. For this project the goal is to be able to give this depth as a hyperparameter. The advantage of this type of ansatz is that it is easy to implement in real NISQ devices. This due to the two qubit gates only interacting with adjacent qubits as seen in figure 1a.

However, it is important to keep the vector space (or Hilbert space) as small as possible. This can be done through alternating block ansatz or ABA. ABA achieves this by first locally entangling the qubits referred to as sub-blocks. Followed up by gradually creating a corralled state by interweaving these sub-blocks This further helps by improving trainability and maintaining high expressibility.

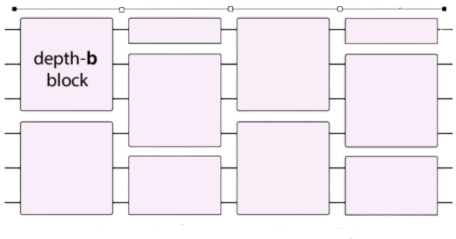
## 4.2 PINNs

Physics Informed Neural Networks or PINNs, are a form of neural network which function as an universal function approximator. The added benefit being the ability to embed any physics law

[1] https://gitlab.fdmci .hva.nl/winsens/pasqal




(a) An example block of the hardware efficient ansatz
**Source:** quantaggle.com/algorithms/ansatz/




(b) A generalized structure of the alternating block ansatz

to prevent impossible solutions from being generated. Even when the law is made up of a partial differential equation. Generally this is achieved by using strict boundary restrictions within the loss calculations. The eventual neural network that this project strives towards will use these characteristics of PINNs to further improve the results generated.

Within the context of a DQC these boundaries work in conjunction with the loss function. Creating extreme loss values when the boundaries are exceeded.

Within the context of DQC the cost function changes the parameter of $\theta$ of a quantum circuit. Using the cost function as an observable defined as a Hermitian cost operator $\hat{C}$. Making it an observable allows for efficient measurement of the scalar function in a NISQ friendly way. This can be represented in the following way where $f_{\varphi,\theta}(x)$ representing optimal boundary states:

$$f(x) = \langle f_{\varphi,\theta}(x)| \hat{C} |f_{\varphi,\theta}(x)\rangle \tag{1}$$

This can be further expanded into several implementation methods.

The most basic form of these boundaries are implemented through the use of Pinned Boundary Handling or PBH for short. The strength of PBH is in the equivalent treatment of boundary and derivative terms. As both of these values are encoded in the eigenspectrum of the cost operators.

The weakness of this method is in the need for extra boundary loss term. Increasing the complexity of calculations and allowing for the competing loss terms.

Floating Boundary Handling or FBH is an alternative to PBH. FBH works through to iteratively shifting the estimated solution based on the boundary or initial point. In this method there is no need for an extra boundary loss term. In stead, it works by iteratively updating the parameterization of the cost function. As the function is parameterized to a specific boundary, the boundary information is worked into the parameters of function.

The advantage of this system is that there is but one loss term. Making sure that the optimizer loss and boundary loss do not start competing. The big disadvantage of this system is that the exact initial starting value for a system.

A final solution is also proposed by the paper [3], Optimized Boundary Handling. It aims to create a hybrid between these two systems. Sadly, the research of this specific method fell out of the scope of this section of the project.

## 4.3 Julia

Scientific computing has traditionally required the highest performance, yet domain experts have largely moved to slower dynamic languages for daily work. There are many good reasons to prefer dynamic languages for these applications, and it is not expected that their use is going to diminish. Fortunately, modern language design and compiler techniques make it possible to mostly eliminate the performance trade-off and provide a single environment productive enough for prototyping and efficient enough for deploying performance-intensive applications. The Julia programming language fills this role: it is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages.

Because Julia's compiler is different from the interpreters used for languages like Python or R, you may find that Julia's performance is unintuitive at first. Once you understand how Julia works, it's easy to write code that's nearly as fast as C.

The most significant departures of Julia from typical dynamic languages are:

- The core language imposes very little; Julia Base and the standard library are written in Julia itself, including primitive operations like integer arithmetic.
- A rich language of types for constructing and describing objects, that can also optionally be used to make type declarations.
- The ability to define function behavior across many combinations of argument types via multiple dispatch.
- Automatic generation of efficient, specialized code for different argument types.
- Good performance, approaching that of statically-compiled languages like C.

Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. In addition to the above, some advantages of Julia over comparable systems include:

- Free and open source (MIT licensed).
- User-defined types are as fast and compact as built-ins.
- No need to vectorize code for performance; devectorized code is fast.
- Designed for parallelism and distributed computation.
- Lightweight "green" threading (coroutines).
- Unobtrusive yet powerful type system.
- Elegant and extensible conversions and promotions for numeric and other types.
- Efficient support for Unicode, including but not limited to UTF-8.
- Call C functions directly (no wrappers or special APIs needed).
- Powerful shell-like capabilities for managing other processes.
- Lisp-like macros and other metaprogramming facilities.

## 4.4 Data preparation

Classical Data must first be applied to a quantum circuit before any operations can be applied to it. While it is possible to simply map a 0-bit to $|0\rangle$ and 1-bit to $|1\rangle$, it is rather wasteful as the information density of a qubit can be much higher.

There are different kinds of data encoding methods that make more use of the properties of a qubit. Some only offering a translation from classical space to the quantum space, with other also directly functioning as a feature map.

In this section of the paper several methods of data encoding are discussed. The primary method of encoding within the project is amplitude embedding and the variants thereof. In the amplitude-embedding technique, data is encoded into the amplitudes of a quantum state. Within this project several versions of amplitude embedding are used and discussed below.

### 4.4.1 Amplitude embedding

The most basic version of Amplitude embedding is the template version supplied by Pennylane. The data is represented by the amplitudes of an n-qubit quantum state where $N = 2^n, x_i$ is the $i$-th element of $x$, and $|i\rangle$ is the $i$-th computational basis state in the following formula.

$$|\psi_x\rangle = \sum_{i=0}^{N} x_i \, |i\rangle$$

However, this is a rather limited to some other options discussed below. Generally, this is because it provides weak feature mapping as well as being limited within the amount of data points by available qubits.

Info on implementing amplitude encoding with Julia/Yao is possibly used here: [4]

### 4.4.2 Feature mapping

But what is feature mapping, and why is it important? Many classical machine learning methods re-express their input data in a different space to make it easier to work with, or because the new space may have some convenient properties.

Support vector machines are a common example, they classify data using a linear hyperplane. A linear hyperplane works well when the data is already linearly separable in the original space, however for many other data sets this is unlikely to be true. To work around this it may be possible to transform the data into a new space where it is linear by way of a feature map.

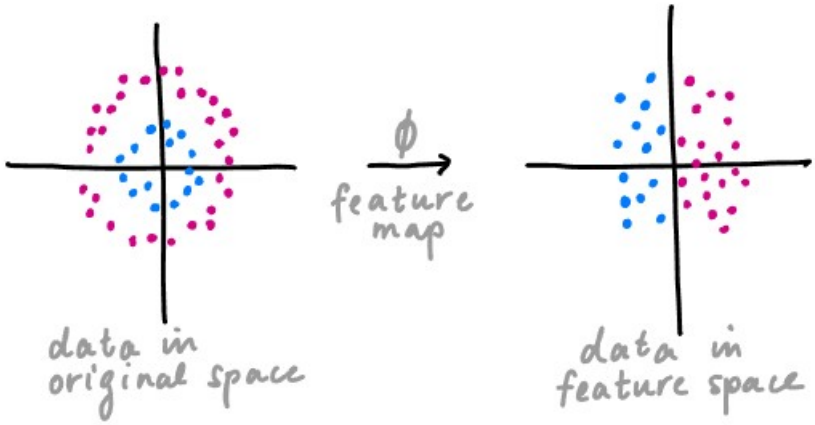



Figure 2: A feature map can transform data into a space where it is easier to process [5].

It takes a classical data point x and translates it into a set of gate parameters in a quantum circuit, creating a quantum state $|\psi_x\rangle$.

Let's consider classical input data consisting of M examples, with $N$ features each,

$$D = x^{(1)}, ..., x^{(m)}, ..., x^{(M)},$$

where $x^{(m)}$ is a $N$-dimensional vector for $m = 1, ..., M$. To embed this data into $n$ quantum subsystems ($n$ qubits or $n$ qumodes for discrete- and continuous-variable quantum computing, respectively), we can use various embedding techniques.

### 4.4.3 Product feature map

The product feature map we consider uses qubit rotations, but has a non-linear dependence on the encoded variable $x$ [3]. An easy example of this is with single layer rotations. For the encoding it could be written in the form

$$\hat{U}_\varphi(x) = \bigotimes_{j=1}^{N'} \hat{R}_{a,j}(\varphi[x]) \tag{2}$$

Where $N' \leq N$ is the number of qubits used for the encoding. $R_{a,j}(\varphi) := exp(-i\frac{\varphi}{2}\hat{P}_{a,j})$ is the Pauli rotation operator. This is for the Pauli matrices $\hat{P}_{a,j} = \hat{X}_j, \hat{Y}_j, or \hat{Z}_j$ where $a = x, y, z$ respectively. While acting on qubit $j$ with phase $\varphi$.

The feature map encodes the data onto the circuit into some quantum state $|\psi_{in}(x_i)\rangle$ by applying a unitary input gate $U(x_i)$.

### 4.4.4 Chebyshev

The Chebyshev feature map expands on this concept by introducing a non-linear transformation upon our data.

Belonging to the product feature map family, it drastically changes the basis set for function representation. As a building block a single qubit rotation $\hat{R}_{y,j}(\varphi[x])$ is used, but with nonlinearity introduced as $2\arccos x$, n = 0,1,2, etc. Unlike the standard product feature map, the Chebyshev map uses Chebyshev polynomials. These Chebyshev polynomials will help the product map with better approximating the function. They help with this because Chebyshev polynomials can take place as various non-linear forms, due to their chaining and nesting properties which make them able to expand very fast. This can be implemented with a change of the phase into $\varphi[x] = 2n[j]\arccos x$ and the Pauli operator into $a = y$ for the Pauli y gate, which changes our equation as follows

$$\hat{U}_\varphi(x) = \bigotimes_{j=1}^{N} \hat{R}_{y,j}(2n[j]\arccos x). \tag{3}$$

Now there are two new variables forming the coefficient $n[j]$, n stands for an arbitrary number starting from zero and j stands for the position of the qubit. In the paper it is written that $n = j$, here n refers to the position of the qubit. This means that the position of the qubit decides the degree of the polynomial.

Chebyshev featuremap makes use of the chaining properties of the polynomials and the encoded degree $n$ will always be equal to one, which gives the following equation

$$\hat{U}_\varphi(x) = \bigotimes_{j=1}^{N} \hat{R}_{y,j}(2\arccos x). \tag{4}$$

### 4.4.5 Chebyshev Tower

Chebyshev polynomials are very useful because of their chaining properties, nesting properties and simple differentiation rule. Within these properties you have the option between two different types of Chebyshev feature maps. The first version is the Chebyshev feature map. The second version is Chebyshev tower feature map, in this version the encoded degree will grow with the amount of qubits. With this method large expressibility can be given without increasing the system size or number of rotations, as seen in the following Equation

$$\hat{U}_\varphi(x) = \bigotimes_{j=1}^{N} \hat{R}_{y,j}(2j\arccos x). \tag{5}$$

# 5 Discussion

In this chapter we will discuss some different aspects of the project. Giving an explanation or justification for certain methods used. This chapter will also be used to talk about the results of the work done in the making of this paper.

## 5.1 Goal

The primary goal of the paper: understand and reproduce the feature mapping in a DQC architecture, has been achieved. The DQC architecture is explained in subsections 4.1 and 4.2. While in Python and in Julia the primary feature maps are available.

The secondary goal: expand the architecture implementations, has been largely achieved. The Python implementation has been expanded to include all the different data encoding methods. While the Julia implementation has been introduced missing only one implementation.

Sadly the secondary goal focussing on expressivity has not been achieved. Simply not enough time was allotted to finish it.

## 5.2 Research

A large part for anyone working on this project will be the research of the different topics. As many of the theories used are still highly experimental with implementations being proprietary, information can be hard to find. The next group working on this project will most likely be focussing on a completely different part of the DQC. This paper alone will most likely not be enough to complete the work. Two further papers are essential for understanding this topic.

1. The original paper [3]
2. The paper of the group before this paper [2]

While this paper tries to introduce the concept of the DQC and ansatz, the primary focus was on the encoding. These two papers go further in depth about the core of these two topics.

## 5.3 Python

The Python implementation is build upon the work of the previous group. However, it has been largely expanded. The implementation still runs on Python with Pennylane, expanded with the Torch machine learning library.

Torch was chosen for several reasons. Torch is relatively simple, which is good as machine learning part was not one of the primary goals of this part of the project. There are also multiple sources already available showing how to use Torch in conjunction with Pennylane. And finally, one of the writers of this report already had some experience with the framework.

All the missing feature maps have been added, and the base of the machine learning has been written. Making it so that at the time of writing it gives a somewhat useless result As it is purely a feature map with a trained bias. However, this is exactly what it is meant to be.

In the future much can still be changed about this groundwork. When implementing the full ansatz (blocks), Torch might no longer be a good library. The training as is currently done on the encoding might be unneeded of even harmful. The code was thus done in such a manner that it should be relatively simple to replace any one component. Choices made here where made with the knowledge of today, any future maintainer, feel free to change them.

## 5.4 Julia

The Julia implementation is built upon a project of SciML, QuantumNLDiffEq [6]. The program uses the YAO package for its quantum processes.

The choice to use julia was taken when the project goals shifted. The paper from Pasqal already introduced yao, and researching in that direction gave a use of julia with the yao package that looked promising. Julia is becoming more accepted and used in the scientific community, because of its properties, as discussed in Section 4.3. Andrea pointed towards a project in development by SciML.

During sprint 3 the SciML program had only a basic implementation of product and chebyshev feature map. An implementation for chebyshev tower was done in the pasqal project. However,

during the last sprint, SciML pushed their own version of the chebyshev tower feature map. Because their implementation was of higher quality, it was decided to merger their variant into the pasqal project. Until the moment of writing the implementation of Amplitude embedding has not been completed. more on that in Section 5.5

### 5.5 Future

For the future of the project, the first part that has to be finished is the failed objective of this part, the expressivity. As briefly described in subsection 5.1, not enough time was allotted to finish it. It is a rather vague topic with several definitions. Instead of trying to explain the concept in a sub par manner, it was decided to remove it from this document to conserve the quality. Sadly, this does mean that it still has to be done.

In the Julia environment 3 feature maps have been implemented, these are the Chebyshev, Chebyshev tower and product feature map. The implementation of Amplitude encoding is not yet usable or fully implemented. The YAO package has multiple ways of creating a circuit. Through Chain's with AbstractBlock's and with ArrayReg's. The Amplitude embedding implementation uses ArrayReg's, while the rest of the program uses AbstractBlock's. These two are as far as known unable to be combined to use with the DQC, therefore the implementation was unable to be tested. The implementation also missed the concept of "Adjusting the coefficients by multiplying the output Yao state by the array of the amplitude coefficients encoding (this should be simply the * operation)", as given by Andrea of the company Pasqal.

The next real part of the project would be in the implementation of the ansatz. Assuming the goal will remain similar in the future, the likely start will be with studying the variational block structure. A big help in understanding this concept was the paper made by the previous group. As it was one of their primary goals to work on. Followed by the implementation of writing routines combining the necessary gates, by having user-defined depth as a hyperparameter. Most likely as an extension of the currently existing ansatz structure in the Python implementation. Including but not limited to the total magnetization of the Z direction, Boundary handling, parameter shift rule, and a cost function. It is at this point unclear what type of ansatz the code of SciML uses, so that needs some research if it is decided to keep using their code. The concept and implementation of the ansatz in Julia should be the same as the Python variant, described in Section 4.1, as the writers of the DQC paper [3] used Julia with the YAO package and described their implementation.

## 6 Conclusions

DQC and in extension PINNs can be extremely useful in the modeling of nonlinear differential equations. However, putting this in practice requires an understanding and an implementation of many highly experimental features.

The DQC or Differentiable Quantum circuit tries to achieve this modeling capabilities by being differentiable. Thus allowing it to be optimized by classical neural network optimizers. Furthermore, it tries to be more efficient by expanding aspects of a PINN or Physics informed Neural network. Which tries to lay strict boundaries to achieve more realistic, and thus usable results. The incorporation of the DQC into the PINN is done through the use variational quantum circuits or ansatz. This effectively replaces the classical neural network with a quantum circuit. Differentiable and thus trainable.

However, this quantum system has no simple ability to take in data from the classical realm. This is where data encoding and feature mapping comes in. While encoding data into a quantum circuit is not particularly difficult, doing it efficiently is. As a qubit has a higher, different dimensional data density as a classical bit, a conversion must be made that is more efficient than 0 to $|0\rangle$ and 1 to $|1\rangle$.

Amplitude encoding is a basic form of this process. However, it is rather limited as it only works if enough qubit capacity is available. It is further limited by its linearity and thus an unsatisfactory

feature mapping ability. Feature mapping attempts to make data more easily usable by a network and thus more efficient.

Encoding methods that are better at this pursuit include but are not limited to the product feature map, Chebyshev and Chebyshev Tower. Each with their own parameters and scale of non-linearity.

These methods can be fully recreated in a classical environment and tested. With different frameworks delivering slightly different results based on their simulation and exact implementation.

# References


[1] *Quantum Startups Pasqal and Qu&Co Announce Merger to Leverage Complementary Solutions for Global Market*. 2022. URL: https://pasqal.io/2022/01/11/quantum-startups-pasqal-and-quco-announce-merger-to-leverage-complementary-solutions-for-global-market/. (accessed: 20.05.2022).

[2] Sebastiaan Groen Danja Verburg and Rohan Webbe. "Solving nonlinear differential equations with differentiable quantum circuits". In: (2021). URL: https://gitlab.fdmci.hva.nl/winsens/pasqal/previous_group/main_report.

[3] Annie Paine Oleksandr Kyriienko and Vincent Elfving. *Solving nonlinear differential equations with differentiable quantum circuits*. Version 2. 2021. arXiv: 2011.10395.

[4] *YAO Basics*. URL: https://yaoquantum.org/notebooks/quick-start/2.yao_basics.html. (accessed: 29.06.2022).

[5] *Quantum Feature Map*. 2019. URL: https://pennylane.ai/qml/glossary/quantum_feature_map.html. (accessed: 03.06.2022).

[6] SciML team. *QuantumNLDiffEq.jl*. Version 912fa6cb45d4ec011567ab87c360c4c626a98bb4, GitLab, june. 22,2022. URL: https://github.com/SciML/QuantumNLDiffEq.jl. (accessed: 28.06.2022).