Project Report

Optimal Quantum Control using High Performance Computing

By

Alexander Pitchford

Document Date: 19 May 2014

Abstract

An implementation of the GRAPE algorithm for optimal quantum control was developed in Python. This provides an open platform, i.e. no license cost, for students and researchers to complete quantum control simulations. The algorithm uses the BFGS or L-BFGS-B method for maximising the fidelity function, and uses the exact gradient method for the Jacobian matrix. The implementation was tested with simulations of the Hadamard and quantum Fourier transform gate synthesis on spin qubit system models. Minimum control drive time optimisation of the central spin model was also completed, which demonstrated that the new implementation may be better at avoiding sub-optimal, local maxima traps. Parallel processing with the HPC-Wales cluster was used, reducing the time required when completing the many simulation runs required to find the global maximum fidelity.

Contents

1 Introduction

The main aim of the project was to implement the GRAPE algorithm for quantum control optimisation on an open platform. This is to allow researchers to run quantum control simulations without the need for licensed software. The project also aimed to add parallel processing to the algorithm, as often many simulation runs are required to find the global optimum, due to optimisations becoming stuck in local maxima. In addition, the project aimed to extend the algorithm to allow control of some infinite dimensional systems to be optimised; specifically these are systems with quadratic Hamiltonians that can be represented by symplectic matrices. These include quantum optics and coupled quantum oscillators, which may provide answers to some of the technical problems inherent in discrete (qubit) systems for quantum computing, and may also be able to perform some tasks more efficiently.

A description of the theory surrounding the project is given in the project dissertation. However, a summary of this is given in section 2, and some areas are expanded where these were found to be important during the project. The primary source for sections 2.1 and 2.2 is Domenico d'Alessandro's book [1], which is widely cited in the quantum control literature. The book by Jorge Nocedal and Stephen J. Wright [2] was used as the main source for section 2.4, as it is cited by the authors of the SciPy optimisation functions.

2 Background Information

2.1 Quantum Control

In order to achieve the state preparation and gate synthesis required for quantum computing it is necessary provide some control over the physical quantum system that is being used to perform the computation. A useful quantum computer would comprise many entities, e.g. spin states, that operate as quantum bits (or qubits). Quantum control looks drive the system as a whole through time varying fields, which act as control pulses, to prepare states and perform gate operations. This offers the potential for quantum computing in systems where physical entities are in fixed locations, e.g. NV centres in diamond [3]. There many other applications for quantum control, such as medical imaging (NMR) and quantum metrology.

The evolution over time of a quantum system is determined by the Hamiltonian, which describes the energy landscape. The system will have its own underlying Hamiltonian (known as the *drift* Hamiltonian) H_d. The effect of an external fields on the energy landscape can be described by Hamiltonians H_j. Where there a number of these, which have amplitudes varying over time according to $u_j(t)$, the time dependent Hamiltonian is given by

$$H(t) = H_d + \sum_{j=1}^{m} u_j(t) H_j \tag{2.1}$$

$u_j(t)$ are known as the *control amplitude functions*.

The evolution over time of the system can be described by an operator $X(t)$ according to Schrödinger's equation (given here in natural units)

$$i\frac{\mathrm{d}}{\mathrm{d}t}X(t) = H(t)X(t), \quad X(0) = \mathbb{1} \tag{2.2}$$

For a finite dimensional system with n dimensions the time evolution of the system can be described by an operator $X(t)$ that is unitary for all times t and $X(t)$ can be represented by an $n \times n$ matrix $U(t)$.

It is possible to determine whether a system is controllable using the Lie rank criterion, as described in detail in [1]. If the desired state can be shown to be reachable for a given H_d and set of H_j, it follows then that there is at least one set of time varying controls that will enable this.

2.2 Optimal Quantum Control

Determining control functions that optimise certain parameters such as the time or energy (or a combination of both) required achieve a target state or evolution to sufficient fidelity is known as *optimal quantum control*. Optimal control theory is well established field and optimal quantum control builds upon this. The cost function is defined with the parameters, including distance from the target, and this is minimised to give the optimal control function.

One of the primary motivations for optimal quantum control simulations is to determine the minimum drive time in which the desired evolution can be achieved, as interactions with the environment lead to loss of purity of quantum state, known as *decoherence*. Hence reducing the time required to achieve the state preparation or gate synthesis will reduce the error in the quantum computation.

Analytical solutions that provide a function of time to describe u_j explicitly are very rare, and only exist for low dimensional systems. A useful quantum computer would have to be many body system, hence high dimensionality, and therefore numerical methods are needed. Many algorithms of this type have been developed and continue to be an active area of research. One of these algorithms is described below.

2.3 The GRAPE Algorithm

The gradient ascent pulse engineering or GRAPE algorithm was introduced in 2005 as a method of determining control pulses for NMR spectroscopy [4]. It is applicable to many quantum systems and hence has been used extensively in research, including at Aberystwyth [5].

The effect of the time varying fields on the quantum system is approximated by discretising the time period into slices where the control fields are constant within that slice. The duration of the time slice, which has to be chosen to be small relative to the dynamics of (2.2), can be variable, but for simplicity here we consider M equal slices of duration Δt. The constant Hamiltonian for that slice can then be calculated using (2.1) . The evolution in that slice is then given by the solution to Schrödinger's equation (2.2)

$$X_k = e^{-iH(t_k)\Delta t} \tag{2.3}$$

where X_k is known as the *propagator* for the k^{th} time slice. The evolution over the total time T can then calculated by

$$X_T = X(T) = X_M \cdot X_{M-1} \cdot \ldots \cdot X_k \cdot \ldots \cdot X_1 \cdot X_0 \tag{2.4}$$

where $X_0 = \mathbb{1}$

The fit of this evolution X_T with the target evolution X_{target} , which could be some quantum gate (unitary operator) for instance, can be determined using the overlap as follows

$$f = \frac{1}{N}\left|\mathrm{tr}\{X_{target}^{\dagger} \cdot X_T\}\right| \tag{2.5}$$

where N is a normalisation constant defined by $N = \left\|X_{target}\right\|_2 = \mathrm{tr}\{X_{target}^{\dagger} \cdot X_{target}\}$, which will be equal to the dimension of the system n for unitary operators. This function f is known as the *fidelity*. $f = 1$ implies that the evolution exactly matches the target operator, whereas $f = 0$ would mean there is no overlap, i.e. they are orthogonal. Note the equation for fidelity differs depending on the type of system and control aim; equation (2.5) gives the fidelity for closed quantum systems (assumed to be isolated from their environment) where the control aim is phase independent. These were the system type and aim used in this project.

The fidelity therefore is a function of the control amplitudes in each time slice, i.e. $f = f(u_{1,1}, \dots, u_{1,k}, \dots, u_{2,1}, \dots, u_{2,k}, \dots, u_{m,M})$. There are m controls and M time slices and hence mM variables. This means that the problem of determining the piecewise control amplitudes that will result in the maximum fidelity can be approached using one of the many standard algorithms for optimising multivariable functions. The simplest of these would be the steepest ascent method. Using an analogy with climbing a hill, i.e. a 2 dimensional system $f = f(x, y)$ where f is the elevation and x and y are the grid coordinates, this is equivalent to walking a step in the steepest direction until there is no longer an upward direction, i.e. the top of the hill. However, it is obvious that this may not be the top of highest hill, it may just be a small hump, or it may be there is a taller peak some miles away. These are known as local maxima. Whilst walking we can use our eyes to see if there are other higher hills near us, and hence head in that direction, however the optimisation algorithms are as if blind or in thick fog, and hence they can only try starting from many different points in the variable space in order to find the global maximum.

All the optimisation algorithms require the gradient w.r.t. each variable to be known or estimated. There are many methods for this, in this project only the *exact gradient* method was used. The calculation for this for closed systems, phase-independent gate synthesis is given in equation (30) of [6]. It was originally derived by S. Schirmer, and is described in [7]. In some quantum systems exact gradients cannot be calculated and hence approximations have to be used.

The steepest ascent is a first order method in that it uses only the first derivative, i.e. the gradient, to determine the iterative step. Second order methods, which use the second derivative, or curvature, are found to converge in fewer iterations than first order methods [2]. One of these, BFGS, is described below in section 2.4.

The GRAPE algorithm allows for all control amplitudes in all timeslots (i.e. all $u_{j,k}$) to be updated in each iterative step. Krotov type methods are similar but look to optimise w.r.t. one time slice at a time. There are hybrids of these methods where subsets of the timeslots are varied. In a comparative study of these methods GRAPE using BFGS was found to provide the best performance, except in some specific examples where Krotov or Hybrid methods were more suitable [6]

2.4 The BFGS Algorithm

The Broyden–Fletcher–Goldfarb–Shanno algorithm is an iterative method for finding the stationary points of multi-dimensional functions. In this section we define the function to be optimised as

$$f = f(x_1, x_2, \dots, x_n) \tag{2.6}$$

and therefore n is the number of variables, or the *dimension* of the function (not to be confused with the n in section 2.1). It is one of class of methods known as *quasi 2nd order Newton* methods.

The name of this class of methods derives from Newton's method for finding the roots of functions. The iterative step for this is given in (2.7). This is equivalent to finding the intersection with the x axis for the tangent to the function, as illustrated in Figure 2-1. The convergence of this method is found to be at least quadratic, whereas with methods not using the first derivative of the function, e.g. the secant method, the convergence is linear. There many caveats to this, that mostly relate to the requirement for the derivative to be well behaved, e.g. non-zero near the root. The method also typically only works well when the initial guess is near the root.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2.7}$$

(a) (b)

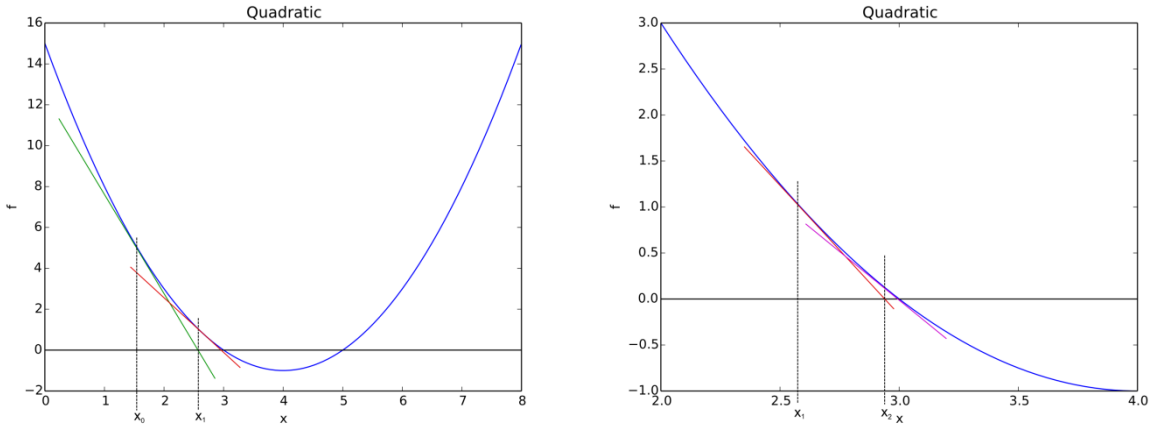

Figure 2-1: Illustration of Newton's method for finding the roots of a function. Starting from an initial guess $x_0 = 1.5$ three iterations find an approximation to the root of 2.997 for the function $f = (x-3)(x-5)$.

When looking for the maxima or minima of a function $f' = 0$, and hence finding the stationary point is finding the roots of f', and hence the iterative step using Newton's method for a single dimension system would be

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{2.8}$$

The second derivative is used to find the stationary points of function, hence these methods are second order. The iterative step in multi-dimensional 2nd order Newton methods use the *Jacobian* and *Hessian* matrices. The Jacobian is a matrix of the first partial derivatives of the function evaluated at a given point, which is the gradient at that point. In the case of f as defined in (2.6) this is a vector of length n, i.e. $\left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]$. The Hessian is the matrix of the second partial derivatives at the given point, which is the local curvature, and in this case is an $n \times n$ matrix. The details of the iterative step are not given here, but can be found in [2].

In most applications, determining the second partial derivatives is either impossible or at least impractical. The BFGS method generates an approximation to the Hessian using the gradients from each iterative step - it actually stores an approximation to the inverse Hessian, as this is what is used in the iterative step. Because it uses an approximation to the Hessian, rather than the Hessian itself, it is referred to as a *quasi* 2nd order Newton method. By using these approximations to the local curvature the method is able to converge on the solution, i.e. the stationary points, in fewer steps

than the steepest ascent method. Note that most BFGS implementations look to find the minima of functions; they can be used to find maxima by negating the function and gradients. The use of the curvature improves the efficiency of finding the local minima, but it does not overcome the problem of knowing whether the local minima is the global minimum, i.e. the algorithm will still get stuck in local minima.

As the Hessian scales with n^2, then for large dimensional problems the overhead of maintaining this matrix can be costly in terms of computer memory (and hence most likely processing time). The limited memory version of BFGS (known as L-BFGS) reduces the memory requirements by storing only pairs of vectors (of length n), that hold the step size and gradient change, for a number of previous steps (typically denoted by m , not to be confused with the m in section 2.1, referred to as the maximum metric correction factor in later sections). The memory requirement is therefore only $2nm$, which is $\ll n^2$ for large n and $m \ll n$. These m vector pairs are used implicitly as the inverse Hessian in the iterative steps. A small value of m e.g. between 3 and 20 should produce satisfactory results according to [2]. If m is greater than the number of iterative steps then the L-BFGS method is equivalent to the BFGS method. BFGS and L-BFGS are unconstrained, meaning there are no boundaries to variable values. The L-BFGS-B method places upper and lower bounds on the variables. This is a popular implementation, as it offers the flexibility of being used as BFGS by setting large m and very large bounds, or used like L-BFGS by setting just very large bounds.

3 Method

The algorithms for the project were written in Python (v2.7.5) using the Spyder (v2.2.4) development environment. The NumPy (v1.7.1) and SciPy (v0.12.0) were used extensively [8] [9]. Matplotlib (v1.3.0) was used to generate plots [10]. The development was undertaken on an ASUS S400C laptop running MS Windows 8.1 (64bit) with an Intel Core I3 processor and 6GB RAM. This was also used to run the shorter simulations. The longer simulations were run on HPC Wales [11]; specifically on the Aberystwyth cluster, which is a Tier 1 cluster with 54 nodes, each with 12 cores (2 Intel Xeon Westmere X5650 processors) and 36GB RAM. The operating system was Red Hat Linux (v5.5) and the Python environment was v2.7.5 with NumPy v1.8.0 and SciPy v0.13.0.

The Python language was chosen because it has been used for other quantum mechanics simulations e.g. QuTiP [12]. It is an increasingly popular choice for scientific programming, as it is free to use, open source and platform independent. The NumPy and SciPy libraries provide an extensive range of mathematical and scientific functions. Specifically for this project this includes eigendecomposition of matrices and BFGS implementations. Python, being an interpreted language, is known to be slow in execution when compared with low level languages, which would make it unsuitable for a numerical optimisation application. However, it is possible to interface with modules written in C or Fortran, and this allows fast executing code to used within the Python environment. An example of this is discussed later in this section.

3.1 GRAPE Algorithm Implementation

As set out in the introduction the main aim of the project was to develop an implementation of the GRAPE on an open platform. The DYNAMO framework can be configured to run a GRAPE algorithm [6]. This was used as a template for the initial development. As the DYMANO framework supports Krotov and hybrid methods it therefore has greater complexity in the source code than is needed for just the GRAPE algorithm. Some of this was not included from the start, i.e. the ability to optimise for a particular subspace of the controls. However, some was included initially, as it was not obvious that it was only needed for Krotov or hybrid methods. Finally, the configuration method was changed from a function pointer style to a sub-classing style. Some of this is discussed in more detail below.

The implementation uses the BFGS (or L-BFGS-B) algorithm to determine the direction in which to change the control values in each time slice. Although these are stored in a $M \times m$ (M being the number of time slots and m being the number of controls) array, the BFGS algorithm simply sees them as $n = Mm$ variables. At each iteration step the fidelity is calculated, if this exceeds the goal, then it terminates. The gradient w.r.t. all the variables is calculated each iteration, which is used to determine the piecewise control amplitudes for the next step. Although a minimum gradient value is passed as a parameter to the BFGS algorithm, this is checked explicitly each step as well. There is a maximum wall (clock) time limit and iteration limit that also that result in termination when exceeded.

The implementation is a Python library called OptQuantCtrl. It has three main classes: 'OptimConfig' that is used to hold the configuration parameters that can then be shared with the other objects; 'TimeSlots' that holds and computes the Hamiltonians, propagators, evolutions and gradients; 'Optimizer' that wraps the optimisation algorithm and applies normalisation. Example scripts that use the library were developed to test the implementation, these are described in section 4.

3.1.1 Matrix exponentiation and propagation

As shown in (2.3) and (2.4), in order to calculate the evolution of X using the piecewise time slices, the propagator for each time slice needs to be calculated by exponentiating the combined Hamiltonian. Therefore, for a given control pulse, the Hamiltonian at each time slice is first calculated using (2.1), then $-iH(t_k)\Delta t$ is exponentiated. The Hamiltonian has to be diagonalised, as for a diagonal matrix D

$$D = \begin{bmatrix} d_{11} & & 0 \\ & \ddots & \\ 0 & & d_{nn} \end{bmatrix}, \qquad e^D = \begin{bmatrix} e^{d_{11}} & & 0 \\ & \ddots & \\ 0 & & e^{d_{nn}} \end{bmatrix} \tag{3.1}$$

whereas for a general square matrix A

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix}, \qquad e^A \neq \begin{bmatrix} e^{a_{11}} & \dots & e^{a_{1n}} \\ \vdots & \ddots & \vdots \\ e^{a_{n1}} & \dots & e^{a_{nn}} \end{bmatrix} \tag{3.2}$$

An eigendecomposition is performed using the numpy.linalg.eig function to yield

$$H = VDV^{\dagger}, \qquad V = [\vec{v}_1 \quad \dots \quad \vec{v}_n], \qquad D = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n) \tag{3.3}$$

i.e V is a square matrix of column eigenvectors $\vec{v}_1, \vec{v}_2, \dots, \vec{v}_n$ and $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues. The propagator is then calculated using

$$X_k = e^{-iH\Delta t} = Ve^{-i\Delta tD}V^{\dagger} \tag{3.4}$$

The forward propagation, that is the evolution operator resulting from driving the system from $t = 0$ up to the time slice t_k , is given by

$$X_{k:0} = X_k \cdot \dots \cdot X_1 \cdot X_0 \tag{3.5}$$

and the onward propagation, that is the evolution operator resulting from driving the system from $t = t_{k+1}$ to $t = T$ that would result in X_{target} , is given by

$$\Lambda^{\dagger}_{M+1:k+1} = X^{\dagger}_{target} \cdot X_M \cdot X_{M-1} \cdot \ldots \cdot X_{k+1} \tag{3.6}$$

These are calculated for all $k = 1,2, \ldots, M$ at each optimisation iteration step. In Krotov and hybrid methods only a subset of the propagator values will change at each step. The DYNAMO framework therefore uses flags to determine which Hamiltonians, and hence which propagators, and hence which evolutions (forward and onward propagation) need recalculating. These are used to reduce the number of exponentiations and matrix multiplications, which can be computationally expensive. This was included initially in the new implementation, but it was found that the BFGS algorithm updated all the time-slot amplitudes at every iteration, and hence there are no performance benefits using these flags with the GRAPE algorithm. These flags were then removed to improve clarity. The code remains part of the OptQuantCtrl library (subclass of TimeSlots), as the library may be extended to Krotov type methods at some point.

3.1.2 Fidelity calculation

The overlap of the evolution operator (pre-normalisation) with the target is used in gradient calculation. It can then by normalised as appropriate to the aim (respecting phase or otherwise) to give the fidelity.

$$f_{\phi_0} = \mathrm{tr}\{X^{\dagger}_{target} \cdot X_T\} \tag{3.7}$$

$$f_{PSU} = \frac{1}{N}\left|f_{\phi_0}\right| \tag{3.8}$$

Where f_{PSU} is fidelity calculation for phase independent evolution as in (2.5). It can be alternatively calculated using (3.9) due to the cyclic equivalence of the trace, i.e. $\mathrm{tr}\{ABC\} = \mathrm{tr}\{CAB\} = \mathrm{tr}\{BCA\}$.

$$f_{\phi_0} = \mathrm{tr}\{\Lambda^{\dagger}_{M+1:k+1} \cdot \mathrm{X}_{k:0}\} \tag{3.9}$$

This is true for any value of k. In Krotov type methods then the timeslot could be chosen to minimise the number of matrix multiplications. Choosing the best k is not trivial though, and as it turned out that in GRAPE (using exact gradients) all the forward and onward evolutions needed to be calculated anyway, the fidelity is calculated using (3.7). The alterative remains part of the library though.

3.1.3 Gradient Calculation

To generate the matrix G of the partial derivatives of the phase independent fidelity w.r.t. the amplitude of each control in each time-slice $u_{j,k}$ (which we will call $g_{j,k}$) a multiple step calculation is required. This is equivalent to equation (30) of [6]. Firstly to clarify the definitions

$$G = \begin{bmatrix} g_{1,1} & \cdots & g_{1,M} \\ \vdots & \ddots & \vdots \\ g_{m,1} & \cdots & g_{m,M} \end{bmatrix}, \qquad g_{j,k} = \frac{\partial f_{PSU}}{\partial u_{j,k}} \tag{3.10}$$

The eigenvectors and values used in the Hamiltonian exponentiation are reused to generate a $n \times n$ factor matrix Q using element-wise operations on the matrices of the eigenvalues R and S.

$$Q = \begin{bmatrix} q_{1,1} & \cdots & q_{n,1} \\ \vdots & \ddots & \vdots \\ q_{n,1} & \cdots & q_{n,n} \end{bmatrix} \quad R = -i\Delta t \begin{bmatrix} \lambda_1 & \cdots & \lambda_1 \\ \vdots & & \vdots \\ \lambda_n & \cdots & \lambda_n \end{bmatrix} \quad S = \begin{bmatrix} e^{-i\Delta t\lambda_1} & \cdots & e^{-i\Delta t\lambda_1} \\ \vdots & & \vdots \\ e^{-i\Delta t\lambda_n} & \cdots & e^{-i\Delta t\lambda_n} \end{bmatrix}$$

$$q_{l,m} = \begin{cases} e^{-i\Delta t\lambda_1} & \text{if } \lambda_l = \lambda_m \\ \dfrac{e^{-i\Delta t\lambda_l} - e^{-i\Delta t\lambda_m}}{-i\Delta t(\lambda_l - \lambda_m)} & \text{if } \lambda_l \neq \lambda_m \end{cases} \tag{3.11}$$

The control Hamiltonian H_j is placed in the basis that diagonalises $H(t_k)$ to give the matrix W, i.e.

$$W = V^{\dagger} H_j V \tag{3.12}$$

$W$is then multiplied element-wise by Q to give the matrix Z, i,e,

$$z_{l,m} = w_{l,m} q_{l,m} \tag{3.13}$$

Z is then returned to the basis of $H(t_k)$ to give the partial derivative of the propagator

$$\frac{\partial X_k}{\partial u_j} = VZV^{\dagger} \tag{3.14}$$

The elements of G are then calculated as

$$g_{j,k} = \frac{\partial f_{PSU}}{\partial u_{j,k}} = \frac{1}{N} \mathrm{Re}\left[f_{\phi_0}^{*} \, \mathrm{tr}\left\{ \Lambda_{M+1:k+1}^{\dagger} \cdot \left(\frac{\partial X_k}{\partial u_j} \right) \cdot \mathrm{X}_{k-1:0} \right\} \right] \tag{3.15}$$

hence the forward and onward propagations are required for all values of k.

The magnitude of the gradient is calculated as

$$|G|^2 = \sum_{j}^{m} \sum_{k}^{M} |g_{j,k}|^2 \tag{3.16}$$

for the purpose of comparing to the minimum gradient that will result in termination of the function optimisation algorithm, i.e. the fidelity is considered converged at a local maximum.

3.1.4 Function Optimisation Algorithms

Two variations of the BFGS function optimisation are implemented in the OptQuantCtrl library. Both are called from the SciPy optimisation module. The first to be implement was the standard BFGS, i.e. unconstrained variables and full Hessian approximation. This was found to perform very slowly when compared with the MATLAB implementation in DYNAMO. In the simulations described in section 4 the processing time was found to increase quadratically with the number of time slices M, which was to expected, as the Hessian matrix is $Mm \times Mm$. The SciPy implementation of BFGS is written in Python, which as mentioned previously, is known to be slower in execution than compiled languages.

The SciPy optimisation module also provides an interface to the L-BFGS-B implementation developed in Fortran by Ciyou Zhu, Richard Byrd, Jorge Nocedal and Jose Luis Morales [13]. The processing time using this algorithm was found to be similar to that of the MATLAB code in DYNAMO. This algorithm

is therefore recommended for all simulations. As described in section 2.4 L-BFGS-B provides the flexibility to act as BFGS, and (especially) in SciPy, with faster processing. It therefore supersedes the other algorithm. Both are available for use as sub classes of the Optimizer.

4 Results and Analysis

The simulations described in this section were completed to test the OptQuantCtrl library. The DYNAMO framework is assumed to produce accurate simulations of quantum control and as such the output of the two frameworks are compared. They were not completed to explore the physics involved and therefore the analysis focuses on the comparison between the outputs.

Natural units are used throughout the library, and hence any times are in Planck time units $t_P \approx 5.4 \times 10^{-44}$ s. Therefore all time values mentioned in this section are given in those units. Length scales or absolute energy values are not considered.

The OptQuantCtrl library, Python scripts, parameter files used, and data produced in the simulations, along with brief instructions, are available to download from:

Windows:
https://dl.dropboxusercontent.com/u/22500717/MPhys_project_source_and_data_AJGP.zip

Linux/Unix:
https://dl.dropboxusercontent.com/u/22500717/MPhys_project_source_and_data_AJGP.tar.gz

The files in the packages are the same, it is just the compression that is different.

4.1 Quantum Fourier Transform Gate

The demonstration script provided with DYNAMO simulates the optimisation of control amplitudes to achieve a QFT gate on a 2 qubit system. The quantum Fourier transform is used in many quantum algorithms, including Shor's algorithm for factoring. For a two qubit system the QFT gate is represented by the following unitary matrix:

$$U_{QFT} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \tag{4.1}$$

The qubits in this case are quantum spins. The effect of magnet fields on spin dynamics are given by the Pauli matrices:

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \qquad \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{4.2}$$

The drift Hamiltonian for the two spin system is:

$$H_d = \frac{1}{2}\{\sigma_x \otimes \sigma_x + \sigma_z \otimes \sigma_z + \sigma_z \otimes \sigma_z\} = \frac{1}{2}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

Four controls are used to drive the system:

$$H_1 = \frac{1}{2}\sigma_x \otimes I = \frac{1}{2}\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$H_2 = \frac{1}{2}\sigma_y \otimes I = \frac{1}{2}\begin{bmatrix} 0 & 0 & -i & 0 \\ 0 & 0 & 0 & -i \\ i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{bmatrix} \tag{4.4}$$

$$H_3 = \frac{1}{2}I \otimes \sigma_x = \frac{1}{2}\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$H_4 = \frac{1}{2}I \otimes \sigma_y = \frac{1}{2}\begin{bmatrix} 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix}$$

where I is the 2×2 identity matrix.

The drive time used was 10. The DYMANO demo suggests 6 is sufficient, but 10 made debugging easier, as the time slices were easier decimals to read. The number of time slices M was set at 500, which gives $\Delta t = 0.02$, which is sufficiently small relative to the dynamics of (4.3). The goal for the fidelity can be defined as:

$$f_{goal} = 1 - \epsilon \tag{4.5}$$

where ϵ is the maximum error. In these simulations $\epsilon = 10^{-6}$. The other termination conditions are not relevant, as the goal is always achieved.

The typical approach to determining piecewise control amplitudes for controlling a system is to start from a pulse generated randomly with a uniform distribution, i.e.

$$u_{j,k} = \alpha R() \tag{4.6}$$

where $R()$ is a function that returns a random real number in the interval $[0,1)$ and α is a scale factor, known as the *initial control scaling*. As there are many possible solutions, i.e. sets of amplitudes that will drive the system as desired, then the simulation could produce a different set of amplitudes for different initial random pulses. This would therefore make it very difficult to compare the output. This also results in rapidly changing control pulses, which also makes qualitative comparison difficult.

(a) (b)

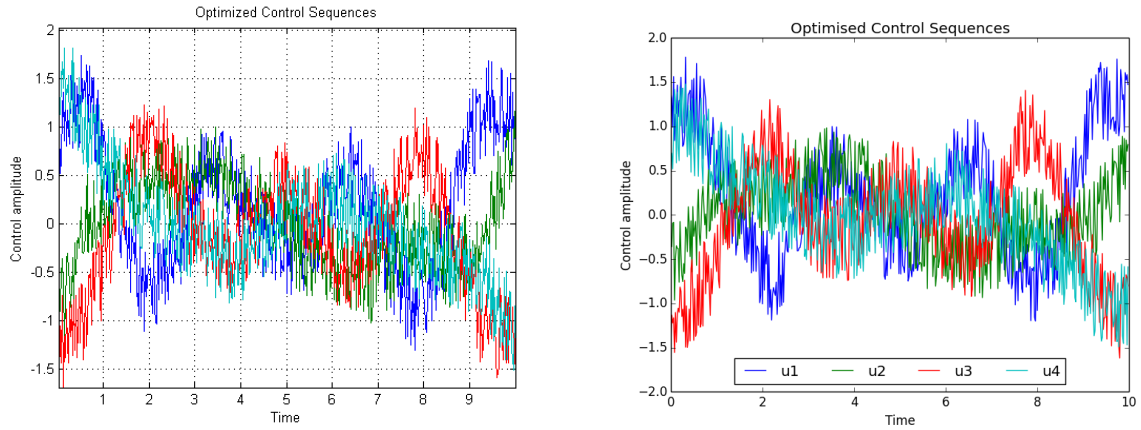

Figure 4-1: Plots showing the control amplitudes found by (a) DYNAMO and (b) OptQuantCtrl (L-BFGS-B) that drive the QFT evolution in $T = 10$ with 500 time slices starting from randomised piecewise control amplitudes

Figure 4-1 does however show a clear similarity in output between the two simulation frameworks. There is the impression of an anti-phase relationship between $u_1$and u_3, similarly with $u_2$and u_4, and generally the same shape to the signal for all the controls.

It was thought that from the same initial pulse both frameworks should find the same control pulses, as they were both using the same optimisation algorithm, and starting from the same point. The control amplitudes were therefore initialised to values in the range [-0.5, 0.5) scaling linearly piecewise from $t = 0$ to $t = T$. As can be seen from Figure 4-2 this is not exactly what was found.

(a) (b)

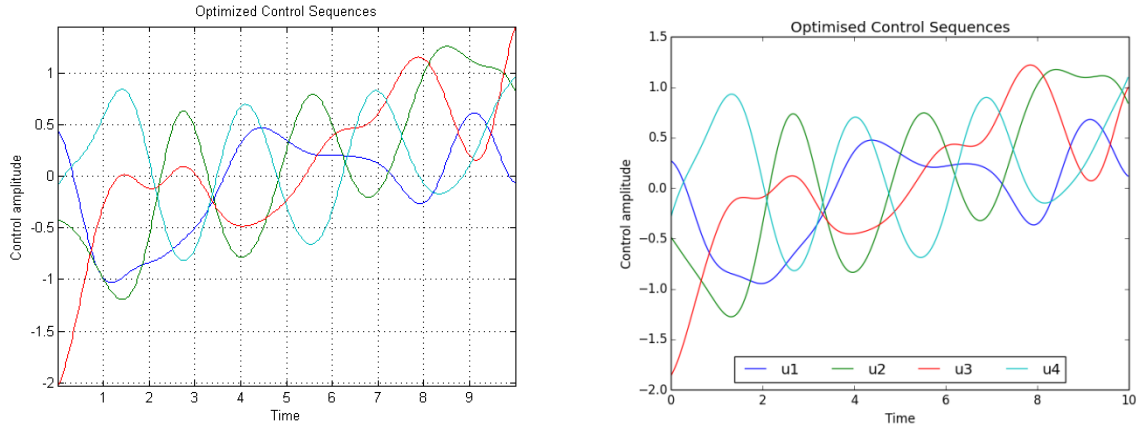

Figure 4-2: Plots showing the control amplitudes found by (a) DYNAMO and (b) OptQuantCtrl (L-BFGS-B) that drive the QFT evolution in $T = 10$ with 500 time slices starting from fixed (linear) piecewise control amplitudes

Although there are clear similarities between the results from the different frameworks, they are not the same control pulses. Attempts were made to try and produce exactly equivalent pulses, such as increasing the fidelity goal and increasing the maximum metric correction factor. These however did not significantly effect on the output from the OptQuantCtrl simulation. It is possible, and in fact assumed to be so, that both sets of pulses would result in the QFT evolution. However, it is not trivial to set up an experiment to test this hypothesis. The possible reasons for the difference in pulse are discussed in section 5.

Figure 4-3 shows an overlay of the output from the three optimisation algorithms used. It shows that each one produces different control pulses, that are all very similar, with L-BFGS-B in OptQuantCtrl and BFGS in DYNAMO being perhaps the most similar. All of these pulses produce the simulated QFT gate to the goal fidelity.

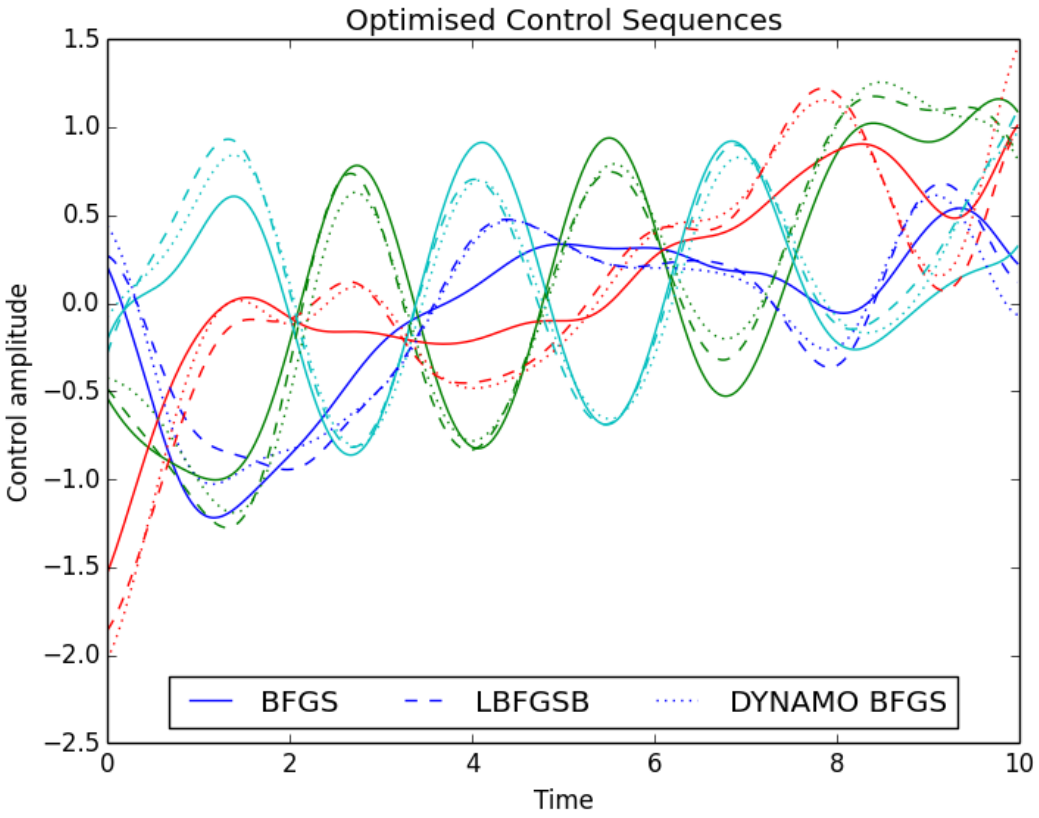

Figure 4-3: Plots showing a comparison of the control pulses produced by three different optimisation algorithms. The solid lines show the results of the SciPy BFGS in OptQuantCtrl, the dash lines SciPy L-BFGS-B in OptQuantCtrl, and the dotted lines are from the MATLAB BFGS (fminopt) in DYNAMO.

4.2 Hadamard Gate

It was found that testing with the most simple system possible was helpful when investigating potential issues with the new numerical framework during its development. The simple, but still relevant, system chosen was the Hadamard gate acting on a single qubit. This is popular choice in quantum information theory, as it forms part of the universal set that perform all unitary transformations. The Hadamard gate acting a single qubit is given by the matrix:

$$U_{Hadamard} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4.7}$$

The drift Hamiltonian that arises from a constant magnetic field in the z direction is given by

$$H_d = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{4.8}$$

and the control being a magnetic field in the x direction, chosen as it does not commute with H_d, is

$$H_1 = \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{4.9}$$

The full Hamiltonian for a given time slice is then

$$H(t_k) = H_d + u_k H_1 \tag{4.10}$$

The drive time used was 10, as in the QFT simulation. The same linear initial pulse was used for the single control amplitude. The number of time slices M was initially set at 200, which gives $\Delta t = 0.05$, and then afterwards at 1000, giving $\Delta t = 0.01$. The goal for the fidelity was set at $\epsilon = 10^{-14}$, which is the maximum feasible with Python for reasons discussed in section 5. Again the other termination conditions are not relevant, as the goal is always achieved.

Figure 4-4 shows the control amplitudes using the three optimisation algorithms. Using $\Delta t = 0.05$ BFGS in DYNAMO (MATLAB) and BFGS in OptQuantCtrl (SciPy) seemed to find the same control pulse, whereas L-BFGS-B in OptQuantCtrl (SciPy)found a pulse that seems to match in terms of character (sine wave with same frequency added to the initial linear trend with t), but a much greater magnitude to the signal. Again, it is possible that both pulses would achieve the Hadamard gate synthesis. When $\Delta t = 0.01$ all three algorithms found the same control pulse. This result perhaps demonstrates the importance of choosing sufficient time slices to ensure Δt is small enough relative to the dynamics.

(a) (b)

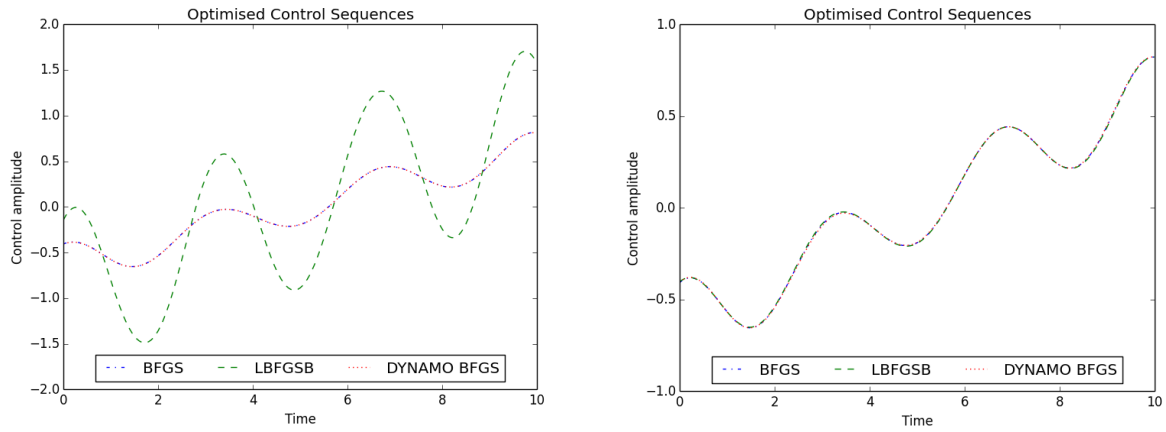

Figure 4-4: Plots of the control amplitudes that drive the Hadamard gate synthesis for a single qubit found by SciPy BFGS and L-BFGS-B in OptQuantCtrl and MATLAB BFGS in DYNAMO. The time slice duration in (a) was $\Delta t = 0.05$ and in (b) $\Delta t = 0.01$

4.3 Spin Star

In order to demonstrate that OptQuantCtrl can be used to produce useful optimal quantum control results, an attempt was made to repeat one to the *spin star* optimisation simulations completed by Christian Arenz et al. [5]. The simulations for this attempt were run on HPC Wales to demonstrate the advantage of using a high performance computing cluster.

4.3.1 The central spin model

The central spin, or spin-star, model simulates the effects of controlling a central spin surrounded by coupled spins, which is a model for the NV centre in diamond coupled with the environmental spins of C^{13} nuclei. This is one of the forerunners for potential physical systems for quantum computation, and is an important area of research in the UK [3]. The model is described in detail in [5], a brief excerpt from this is given below.

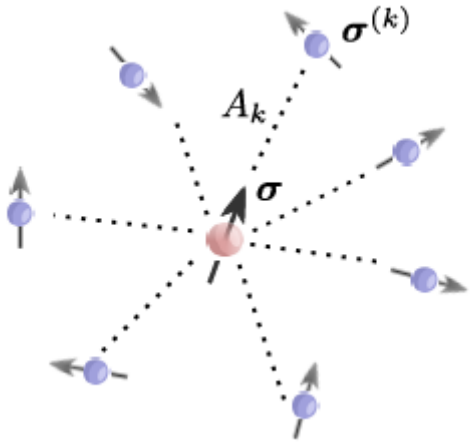

Figure 4-5: An illustration of the central spin model. The Hamiltonian of the central spin is described by $\vec{\sigma}$ and the surrounding N bath spins by $\vec{\sigma}^{(k)}$. The magnitude of each coupling is given by the constant A_k. (diagram taken from [5])

The central spin is assumed to interact with the surrounding bath spins via an isotropic Heisenberg interaction. The overall drift Hamiltonian is given by:

$$H_d = \sigma_y + \sum_{k=1}^{N} A_k \, \vec{\sigma} \cdot \vec{\sigma}^{(k)} \tag{4.11}$$

where

$$\vec{\sigma} = [\sigma_x \quad \sigma_y \quad \sigma_z], \qquad \vec{\sigma}^{(k)} = \begin{bmatrix} \sigma_x^{(k)} \\ \sigma_y^{(k)} \\ \sigma_z^{(k)} \end{bmatrix}$$

are the Pauli matrices acting on the central and bath spins respectively.

If all the coupling constants A_k are equal, then the system can be modelled with a two particle Hamiltonian.

$$H_d = i(\sigma_- - \sigma_+) + 2A(\sigma_- J_+ + J_- \sigma_+ + \sigma_z J_z) \tag{4.12}$$

where $\sigma_\pm = (\sigma_x \pm i\sigma_y)/2$ are the lowering and raising operators acting on the central spin. The bath spins are modelled as a single particle with angular momentum

$$\vec{J} = \sum_{k=1}^{N} \vec{\sigma}^{(k)} \tag{4.13}$$

the raising and lowering operators in (4.12) are given by $J_\pm = J_x \pm iJ_y$ where $\vec{J} = [J_x \quad J_y \quad J_z\,]^T$.

The alternative Hamiltonian (4.12) reduces the dimension of the system for $N > 2$, and hence the size of the Hilbert space, which reduces the processing cost of the simulation. It would be practically impossible to simulate the higher N values using the original Hamiltonian (4.11). The dimension of

the Hilbert space for the original Hamiltonian is 2^{N+1}, whereas the dimension of the Hilbert space for the alternative Hamiltonian is given by

$$d = \begin{cases} \frac{1}{2}(N+2)^2 & \text{for even } N \\ \frac{1}{2}(N^2+4N+3) & \text{for odd } N \end{cases} \tag{4.14}$$

The Hilbert space dimension for the simulations completed is given in Table 4-1

The control, a magnetic field in the z direction, is assumed to only act on the central spin, and is modelled by

$$H_c(t) = B(t)H_1 = B(t)\sigma_z \tag{2.1}$$

The full Hamiltonian is therefore

$$H(t) = H_d + H_c(t) \tag{2.2}$$

4.3.2 Quantum control optimisation

The aim of the simulations completed was to determine the shortest time T_{g1} required to achieve some control over the central spin. The control attempted was synthesis of Hadamard gate acting on the central spin, where the couplings to the bath spins are equal. The shortest time is a typical example of a quantum control optimisation objective, for reasons described in section 2.2.

The strategy was to run repeated simulations with increasing total drive (or evolution) time T. For each T a number of attempts is made to reach the target fidelity. If the fidelity goal is achieved, then no further attempts are made for that T. A maximum number of attempts is made, after which the highest fidelity found for that T is recorded. Optionally, the amplitudes that achieve the maximum fidelity can also be saved to file. The search starts at $T = T_{start}$ and continues until $T = T_{stop}$ in increments of ΔT. The search continues for all the values of T even when the target fidelity is reached for some $T = T_{g1}$, as this does not imply that it will be achieved for all $T > T_{g1}$. This process was repeated for the number of bath spins $N = 0, 1, \dots, 7$.

As the main aim of this exercise was to compare the results and performance of the two simulation frameworks, the parameters used in the OptQuantCtrl simulation were those used by Arenz in his DYNAMO simulations. The time slice duration was set at $\Delta t = 0.05$ for all values of T, hence the number of timeslots M was calculated for each T. The driving time limit $T_{stop} = 10$ was used for all N. The Arenz simulations set $\Delta T = 0.1$, however it was felt that $\Delta T = 0.2$ should provide a sufficient comparison, as this does change the physics, it just halves the number of data points, and hence the processing time required. Generally $T_{start} = \Delta T$. The maximum number of attempts for a given T was 200, wherever possible. In some cases this was lowered, i.e. where is was unnecessary (N < 2) as the maximum fidelity was found with typically fewer attempts, or when results were required in a shorter time. The L-BFGS-B algorithm was used for all simulations. The target fidelity was set at 0.99 (i.e. $\epsilon = 0.01$) this is less than used by Arenz (0.995), but as only a visual comparison of plots will be made in the analysis, then this would be negligible.

The calculation of the drift Hamiltonian (both original and alternative) is quite involved, and is therefore not described here. It can be found in the source code (spinstarconfig.gen_full and spinstarconfig.gen_reduced). The control Hamiltonian is calculated by

$$H_1 = \sigma_z \otimes I \tag{2.3}$$

where I is the $(d/2) \times (d/2)$ identity matrix. Note this is not normalised, however this simply equivalent to setting the initial control scaling to d.

The full Hamiltonian in a given time slice is therefore

$$H(t_k) = H_d + u_{1,k} H_1 \tag{2.4}$$

The target is calculated by

$$U_{target} = U_{Hadamard} \otimes I \tag{2.5}$$

Table 4-1: Hilbert space dimension for simulations

N	Hilbert Space Dimension d	
	Original H_d	Alternative H_d
0	2	2
1	4	4
2	8	8
3	16	12
4	32	18
5	64	24
6	128	32
7	256	40

The simulations were first attempted using the original Hamiltonian. However, it was found that the results for the $N = 2$ configuration did not closely resemble those produced by Arenz (using the alternative Hamiltonian). The goal was not achieved for any of the evolution times T tried, whereas in the Arenz simulation this was achieved for $T > \sim 4.5$ in most cases. Assuming that the function optimisation algorithm (L-BFGS-B) was failing to find the global maximum, i.e. it was continually becoming stuck in the same local maxima, attempts were made to find parameters for the algorithm that would avoid this.

The algorithm was found to be terminating due to the maximum iteration limit (250) in most cases where $T > 2.2$. An attempt was made with the limit at 1000, but this was still reached for most T. Converging (i.e. gradient normal minimum reached) before the iteration limit is reached reduces the processing time. The BFGS algorithm is unconstrained, i.e. there are no bounds to the variables, whereas the L-BFGS-B algorithm is bounded. The control values for the pulse that produced the maximum fidelity were found to have reached the bounds (defaulted to ± 100) for some timeslots. Changing these to $\pm 10^6$ resulted in more convergences.

A major improvement came from increasing the maximum number of metric corrections m_{corr} from 10 to 500, which as described in section 2.4 forces the L-BFGS algorithm to behave as the BFGS. This led to the fidelity converging on a maximum, for a given set of starting amplitudes, in typically less than 250 iterations. This demonstrates the advantages of including (or in this case improving) the second order approximation when searching for local minima. Increasing the convergence rate, i.e. reducing the number of iterations required to find local minima, reduces the processing time required to try and locate the global minimum. There is a compromise however, in that the cost of each iteration will increase with m_{corr} . Using a value of $m_{corr} = 100$ was found to give similar iteration counts to $m_{corr} = 500$, but at a much reduced (less than half) overall processing time. This has not been exhaustively tested, and hence this will probably not be the optimal value.

The minimum gradient was decreased from initial guess value of 10^{-8} to 10^{-10}. It was thought that this may help escape what seemed to be local maxima, but may have only been a shallow saddle. On a short experimental run this did find any new maxima, but did lead to an increase in number of iterations. This is not by any means conclusive, but did indicate that it would be better to use the processing time trying different starting points. The minimum gradient value of 10^{-7} was used for all the remaining simulations.

It is assumed that there must be some issue with the configuration for the original Hamiltonian, possibly the value for A (the coupling constant). It was also not possible to find a control pulse that achieved the fidelity goal using the DYNAMO simulation with $N = 2, T = 10$ with the original Hamiltonian after 200 attempts. A switch was therefore made to using the alternative Hamiltonian, as this is what Arenz used to produce his results. The rest of the simulation parameters remained unchanged.

As the simulations were run on HPC Wales, the process of searching for T_g for each N could be run in parallel with negligible performance degradation on an individual process. The processes searching for T_g for large N (> 4) when $T > 5$ were found to be very costly, and hence it was necessary to further split the task into $T_{start} = 5.2, T_{stop} = 6.0$ type searches, i.e. five processes working in parallel to complete the simulations for $T > 5$ for $N = 5, 6$ and 7. The number of attempts was also reduced to 100 for these searches to reduce the processing time required.

(a) (b)

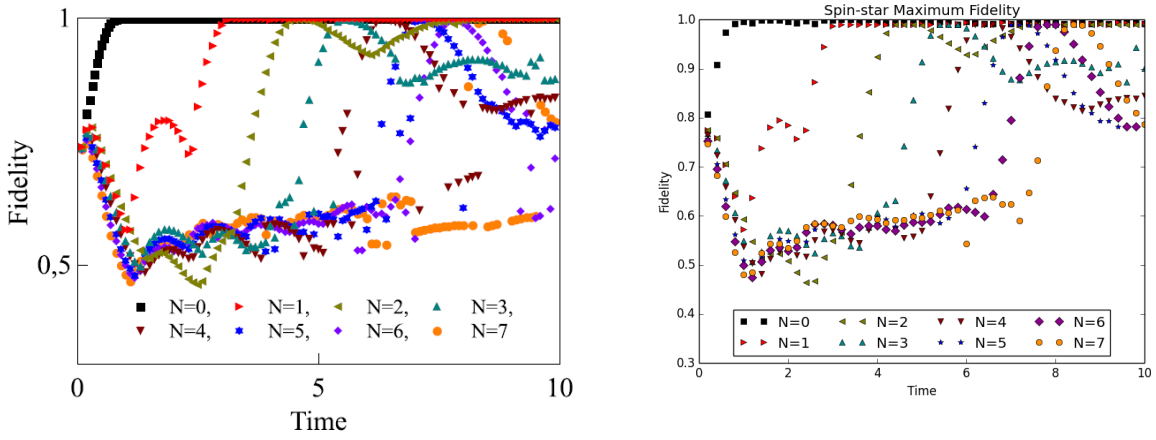

Figure 4-6: Plots showing the maximum fidelity found by (a) DYNAMO and (b) OptQuantCtrl for the spin star model with the alternative (reduced Hilbert space) Hamiltonian

The plots in Figure 4-6 show the comparison between the fidelities found by Arenz with the DYNAMO framework and those found using the OptQuantCtrl library. It is clear that OptQuantCtrl achieved the same result as DYNAMO, with the goal being reached for some $T < 10$ for all N. It

would also seem that OptQuantCtrl was less likely to get stuck in local maxima, as there are fewer data points that fall outside of the general trend - most notable for $N = 4$ and $N = 7$ where $T > 7$. This is despite the fact that fewer attempts from different initial starting points were made.

5 Discussion

The OptQuantCtrl framework has been demonstrated to comparable to DYNAMO in the subset of the DYNAMO functionality that it provides in terms of outputs and execution speed. The small differences seen in the control amplitudes found in the simulations most probably arise from the differences in precision of floating point numbers between Python (53 bit) and MATLAB (64 bit), and differences in the methods used by the function optimisation algorithms. The standard MATLAB BFGS algorithm uses the trust region method (according to the MATLAB documentation for fminopt) whereas the SciPy L-BFGS-B version is an interface to the Zhu, Byrd, and Nocedal implementation that uses a line search method [13]. It would be possible to further compare OptQuantCtrl with DYNAMO package by replacing the call to the BFGS algorithm with the Zhu, Byrd, and Nocedal L-BFGS-B implementation. A MATLAB interface to this is available [14].

Using parallelisation on a high performance computing cluster allowed results to be generated more quickly. It is believed that it took multiple weeks to generate the results for the Hadamard control of the central spin model described in section 4.3. Using simple parallel processing on the HPC-Wales cluster, it was possible to generate the results in two days. This is not an entirely fair comparison, as some comprises were made, but is indicative. There are opportunities to further reduce the time waiting for results; only a small amount of HPC-Wales resource was used. There were no more than 50 cores working on the simulations at any one time. The manual nature of the parallelisation configuration and collation of results was prohibitive to using more parallel processes. HPC-Wales has thousands of cores in total, however there is of course competition for the resource.

5.1 Further Work

Automated parallelisation of the simulation runs could easily be achieved using a Python implementation of the Message Passing Interface (MPI) . MPI is the standard for communicating between processes on HPC clusters. There are MPI interfaces in Python available in the 'pyMPI' and 'mpi4py' libraries. However, neither of these are currently available on HPC-Wales, although this is being addressed. Using MPI, the many attempts achieve goal fidelities could be distributed across all available processors and then collated and compared within the code to determine the highest fidelity found by all processes.

The NumPy library is already multithreaded, and this was found to greatly improve the efficiency of the exact gradient calculation running on HPC-Wales compared with running it on a laptop. Multi-threading could fairly easily be added explicitly to the code, which would reduce simulation time on multi-core processor machines. It would not offer any improvement on HPC clusters, where multiprocessing is a better option, as described above.

Other parts of the functionality of DYNAMO could be added to the library, such as normalisation for phase respecting evolution, fidelities for open systems, gradient functions, and Krotov type methods. Some of these are more trivial than others, and there are no immediate drivers for them. Extension of the library to support infinite dimensional systems with quadratic Hamiltonians using symplectic matrices is however of immediate interest, as this is not available in the DYNAMO framework or elsewhere. Progress has been made by Daniel Burgarth, building on his work in [15], in deriving the equations for the exponentiation and for the gradient.

The algorithm could be made more efficient by using a low level language such as C/C++ or Fortran to complete the processor intensive operations such as the exponentiations and gradient calculations. The disadvantage of this would be that parts of the library would need to be compiled

in a system specific way. A fully C++ version could be developed using a linear algebra library such as ALGLIB [16]. This would again be more difficult to install and the code would be less accessible.

Alternative methods for producing randomised initial pulses may result in faster convergence, and may improve the quality of control pulse. As has been seen in this project, the starting values of the piecewise control amplitudes have a significant effect on the final control pulse. The pulses found by the simulations that started from linear amplitudes were smooth curves, whereas those starting from random amplitudes were erratic. It should be easier to produce smooth signals for use in some physical experiment. Random walks through the time slices would give randomised continuous starting conditions. Or something similar to a Fourier series with random coefficients would also give a initial signal that was continuous and smooth, these could then be approximated piecewise to give initial values for the amplitudes in the time slices.

6 Conclusions

The OptQuantCtrl python library and example scripts provide a framework for simulating finite dimensional quantum systems. This has been demonstrated for qubit systems based on quantum spins. The usefulness of the library in optimal quantum control has been demonstrated. The results of the simulations were found to be comparable with DYNAMO (MATLAB), which is an established and respected frame for quantum control simulation. The tests also showed that the L-BFGS-B algorithm used in OptQuantCtrl may better at finding the global maximum than the MATLAB BFGS algorithm in DYNAMO. The OptQuantCtrl library allows simulations to be completed in less time by parallelising processing on high-performance clusters, which would not be possible on HPC-Wales using DYNAMO due to MATLAB licensing constraints.

There are many opportunities to improve the efficiency and extend the functionality of the library. Many of these will be fairly straightforward to achieve, e.g. automating parallelisation.

7 Acknowledgements

I would like to thank the HPC-Wales team for allowing the use of the Aberystwyth cluster in this project, without which it would not have been possible to produce the results in the time available. Thanks specifically to Sid Kashyap for running weekly working sessions where the theoretical and practical aspects of HPC Wales were explained, and for providing direct support for using Python on HPC-Wales.

8 References

[1] d'Alessandro, D. *Introduction to quantum control and dynamics*. (Chapman & Hall/CRC, 2008).

[2] Jorge Nocedal and Stephen J. Wright, *Numerical Optimization.* (Springer 1999)

[3] Zhao, N. *et al.* Sensing single remote nuclear spins. *Nat. Nanotechnol.* **7,** 657–662 (2012).

[4] Khaneja, N., Reiss, T., Kehlet, C., Schulte-Herbruggen, T. & Glaser, S. J. Optimal control of coupled spin dynamics: Design of NMR pulse sequences by gradient ascent algorithms. *J. Magn. Reson.* **172,** 296–305 (2005).

[5] Arenz, C., Gualdi, G. & Burgarth, D. Control of open quantum systems: Case study of the central spin model. *arXiv* **1312.0160** (2013)

[6] Machnes, S. *et al.* Comparing, Optimising and Benchmarking Quantum Control Algorithms in a Unifying Programming Framework. *Phys. Rev. A* **84,** 1–23 (2010).

[7] Schirmer, S. G. & Fouquieres, P. De. Efficient algorithms for optimal control of quantum dynamics: the Krotov method unencumbered. *New J. Phys.* **13,** 073029 (2011).

[8] many authors. NumPy: The fundamental package for scientific computing with Python (c1995). http://www.numpy.org/

[9] Eric Jones, Travis Oliphant, Pearu Peterson and others. SciPy: Open Source Scientific Tools for Python (2001). http://www.scipy.org/

[10] Hunter, J. D. Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.* **9,** 90–95 (2007).

[11] Fujitsu, HEFCW, WEFO. HPC Wales: An innovative collaboration which gives businesses and researchers access to world-class high performance computing technology. http://www.hpcwales.co.uk/

[12] J. R. Johansson, P. D. Nation, and F. Nori: "QuTiP 2: A Python framework for the dynamics of open quantum systems.", *Comp. Phys. Comm*. **184**, 1234 (2013) [DOI:10.1016/j.cpc.2012.11.019].

[13] Byrd, R. H., Lu, P., Nocedal, J. & Zhu, C. A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM J. Sci. Comput.* **16,** 1190–1208 (1995).

[14] Stephen Becker. MATLAB interface to the L-BFGS-B implementation (2012). http://www.mathworks.com/matlabcentral/fileexchange/35104-lbfgsb--l-bfgs-b--mex-wrapper

[15] Genoni, M., Serafini, A., Kim, M. & Burgarth, D. Dynamical Recurrence and the Quantum Control of Coupled Oscillators. *Phys. Rev. Lett.* **108,** 1–5 (2012).

[16] ALGLIB: a cross-platform numerical analysis and data processing. http://www.alglib.net/