EPSRC Centre for Doctoral Training
Quantum Engineering

University of BRISTOL

Doctorate of Philosophy

Schrödinger's Catwalk

Brian Flynn

University of Bristol

February, 2021

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

ACRONYMS

C	carbon
^{14}N	nitrogen-14
AI	artificial intelligence
AIC	Akaike information criterion
AICC	Akaike information criterion corrected
BF	Bayes factor
BFEER	Bayes factor enhanced Elo ratings
BIC	Bayesian information criterion
CLE	classical likelihood estimation
CPU	central processing unit
DAG	directed acyclic graph
EDH	experiment design heuristic
ES	exploration strategy
ET	exploration tree
FH	Fermi-Hubbard
FN	false negatives
FP	false positives
GA	genetic algorithm
GES	genetic exploration strategy
GPU	graphics processing unit
HPD	high particle density
IQLE	interactive quantum likelihood estimation
JWT	Jordan Wigner transformation
LE	Loschmidt echo

LTL	log total likelihood
ML	machine learning
MVEE	minimum volume enclosing ellipsoid
MW	microwave
NN	neural network
NV	nitrogen-vacancy
NVC	nitrogen-vacancy centre
OF	objective function
PBS	portable batch system
PGH	particle guess heuristic
PL	photoluminescence
QC	quantum computer
QHL	quantum Hamiltonian learning
QL	quadratic loss
QLE	quantum likelihood estimation
QM	quantum mechanics
QML	quantum machine learning
QMLA	Quantum Model Learning Agent
SMC	sequential monte carlo
SVM	support vector machine
TLTL	total log total likelihood
TN	true negatives
TP	true positives
VQE	variational quantum eigensolver

GLOSSARY

Q	Quantum system which is the target of Quantum Model Learning Agent, i.e. the system to be characterised
champion model	The model deemed by Quantum Model Learning Agent (QMLA) as the most suitable for describing the target system
chromosome	A single candidate, in the space of valid solutions to the posed problem in a genetic algorithm
expectation value	Average outcome expected by measuring an observable of a quantum system many times, **??**
gene	Individual element within a chromosome
hyperparameter	Variable within an algorithm that determines how the algorithm itself proceeds
instance	A single implementation of the QMLA algorithm, resulting in a nominated champion model
likelihood	Value that represents how likely a hypothesis is. Usually used in the context of likelihood estiamation, Section 1.3
model	The mathematical description of some quantum system, **??**
model space	Abstract space containing all descriptions (within defined constraints such as dimension) of the system as models
probe	Input probe state, $\|\psi\rangle$, which the target system is initialised to, before unitary evolution
results directory	Directory to which the data and analysis for a given run of QMLA are stored
run	Collection of QMLA instances, usually targeting the same system with the same initial conditions

spawn	Process by which new models are generated, ususally by combining previously considered models
success rate	Fraction of instances within a run where QMLA nominates the true model as champion
term	Individual constituent of a model, e.g. a single operator within a sum of operators, which in total describe a Hamiltonian.
volume	Volume of a parameter distribution's credible region, Section 1.5.1
win rate	For a given candidate model, the fraction of instances within a run which nominated it as champion

Part I

ALGORITHMS

1 QUANTUM HAMILTONIAN LEARNING

First suggested in [1] and since developed [2, 3] and implemented [4, 5], quantum Hamiltonian learning (QHL) is a machine learning algorithm for the optimisation of a given Hamiltonian parameterisation against a quantum system whose model is known a priori. Given a target quantum system, Q, known to be described by some Hamiltonian $\hat{H}(\vec{\alpha})$, QHL optimises $\vec{\alpha}$. This is achieved by interrogating Q and comparing its outputs against proposals $\vec{\alpha}_p$. In particular, an experiment is designed, consisting of an input state, $|\psi\rangle$, and an evolution time, t. This experiment is performed on Q, whereupon its measurement yields the datum $d \in \{0, 1\}$ – i.e. the eigenstate $|d\rangle \in \{|0\rangle, |1\rangle$ is observed – according to the expectation value $\left| \langle\psi| e^{-i\hat{H}_0 t} |\psi\rangle \right|^2$. Then, on a trusted (quantum) simulator, proposed parameters $\vec{\alpha}_p$ are encoded to the known Hamiltonian, and the same probe state is evolved for the chosen t and projected on to $|d\rangle$, i.e. $\left| \langle d| e^{-i\hat{H}(\vec{\alpha}_p)t} |\psi\rangle \right|^2$ is computed. The task for QHL is then to find $\vec{\alpha}'$ for which this quantity is close to 1 for all values of $(|\psi\rangle, t)$, i.e. the parameters input to the simulation produce dynamics consistent with those measured from Q.

The procedure is as follows. A *prior* probability distribution $\Pr(\vec{\alpha})$ in a parameter space of dimension $|\vec{\alpha}|$ is initialised to represent the constituent parameters of $\vec{\alpha}$. $\Pr(\vec{\alpha})$ is typically a multivariate normal (Gaussian) distribution; it is therefore necessary to pre-suppose some mean and width for each parameter in $\vec{\alpha}$. This imposes prior knowledge on the algorithm whereby the programmer must decide the range in which parameters are *likely* to fit: although QHL is generally robust and capable of finding parameters outside of this prior, the prior must at least capture the order of magnitude of the target parameters. It is important to understand, then, that QHL removes the prior knowledge of precisely the parameter representing an interaction in Q, but does rely on a ball-park estimate thereof from which to start.

In short, QHL samples parameter vectors $\vec{\alpha}_p$ from $\Pr(\vec{\alpha})$, simulates experiments by computing the *likelihood* $\left| \langle d| e^{-i\hat{H}(\vec{\alpha}_p)t} |\psi\rangle \right|^2$ for experiments $(|\psi\rangle, t)$ designed by a QHL heuristic subroutine, and iteratively improves the probability distribution of the parameterisation $\Pr(\vec{\alpha})$ through standard *Bayesian inference*. A given set of $(|\psi\rangle, t)$ is called an *experiment*, since it corresponds to preparing, evolving and measuring Q once[1]. QHL iterates for N_e experiments. The parameter vectors sampled are called *particles*: there are N_p particles used per experiment. Each particle used incurs one further calculation of the likelihood function – this calculation, on a classical computer, is exponential in the number of qubits of the model under consideration (because each unitary evolution relies on the exponential of the $2^n \times 2^n$ Hamiltonian matrix of n qubits). Likewise, each additional experiment incurs the cost of calculation of N_p particles, so the total

1 Experimentally, this may involve repeating a measurement many times to determine a majority result and to mitigate noise.

cost of running QHL to train a model is $\propto N_e N_p$. It is therefore preferable to use as few particles and experiments as possible, though it is important to include sufficient resources that the parameter estimates have the opportunity to converge. Access to a fully operational, trusted quantum simulator admits an exponential speedup by simulating the unitary evolution instead of computing the matrix exponential classically.

1.1 BAYES RULE

Bayes' rule is used to update a probability distribution describing hypotheses, $\Pr(\text{hypothesis})$, when presented with new information (data). That is, the probabilty that a hypothesis is true is replaced by the initial probability that is was true, $\Pr(\text{hypothesis})$, multiplied by the likelihood that the new data would be observed were that hypothesis true, $\Pr(\text{data}|\text{hypothesis})$, normalsied by the probability of observing that data in the first place, $\Pr(\text{data})$. It is stated as

$$\Pr(\text{hypothesis}|\text{data}) = \frac{\Pr(\text{data}|\text{hypothesis}) \times \Pr(\text{hypothesis})}{\Pr(\text{data})}. \tag{1.1}$$

We wish to represent our knowledge of Hamiltonian parameters with a distribution, $\Pr(\vec{\alpha})$: in this case hypotheses $\vec{\alpha}$ attempt to describe data, $\mathcal{D}$, measured from the target quantum system, from a set of experiments $\mathcal{E}$, so we can rewrite Bayes' rule as

$$\Pr(\vec{\alpha}|\mathcal{D};\mathcal{E}) = \frac{\Pr(\mathcal{D}|\vec{\alpha};\mathcal{E})\ \Pr(\vec{\alpha})}{\Pr(\mathcal{D}|\mathcal{E})}. \tag{1.2}$$

We can consider Eq. (1.2) at the level of single *particles* (individual vectors in the parameter space), sampled from $\Pr(\vec{\alpha})$:

$$\Pr(\vec{\alpha}_p|d;e) = \frac{\Pr(d|\ \vec{\alpha}_p;\ e)\ \Pr(\vec{\alpha}_p)}{\Pr(d|e)} \tag{1.3}$$

where

- e are the experimental controls of a single experiment, e.g. evolution time and input probe state;
- d is the datum, i.e. the (usually) binary outcome of measuring Q under conditions e;
- $\vec{\alpha}_p$ is the *hypothesis*, i.e. a single parameter vector, called a particle, sampled from $\Pr(\vec{\alpha})$;
- $\Pr(\vec{\alpha}_p|d;e)$ is the *updated* probability of this particle following the experiment e, i.e. accounting for new datum d, the probability that $\vec{\alpha} = \vec{\alpha}_0$;
- $\Pr(d|\vec{\alpha}_p;e)$ is the likelihood function, i.e how likely it is to have measured the datum d from the system assuming $\vec{\alpha}_p$ are the true parameters and the experiment e was performed;
- $\Pr(\vec{\alpha}_p)$ is the probability that $\vec{\alpha}_p = \vec{\alpha}_0$ according to the prior distribution $\Pr(\vec{\alpha})$, which we can immediately access;

- $\Pr(d|e)$ is a normalisation factor, the chance of observing d from experiment e irrespective of the underlying hypothesism such that $\sum_{\{d\}} \Pr(d|e) = 1$.

In order to compute the updated probability for a given particle, then, all that is required is a value for the likelihood function. This is equivalent to the expectation value of projecting $|\psi\rangle$ onto $|d\rangle$, after evolving $\hat{H}(\vec{\alpha}_p)$ for t, i.e.

$$\Pr(d|\vec{\alpha};e) = \left|\langle d|\, e^{-i\hat{H}(\vec{\alpha}_p)t}\, |\psi\rangle\right|^2, \tag{1.4}$$

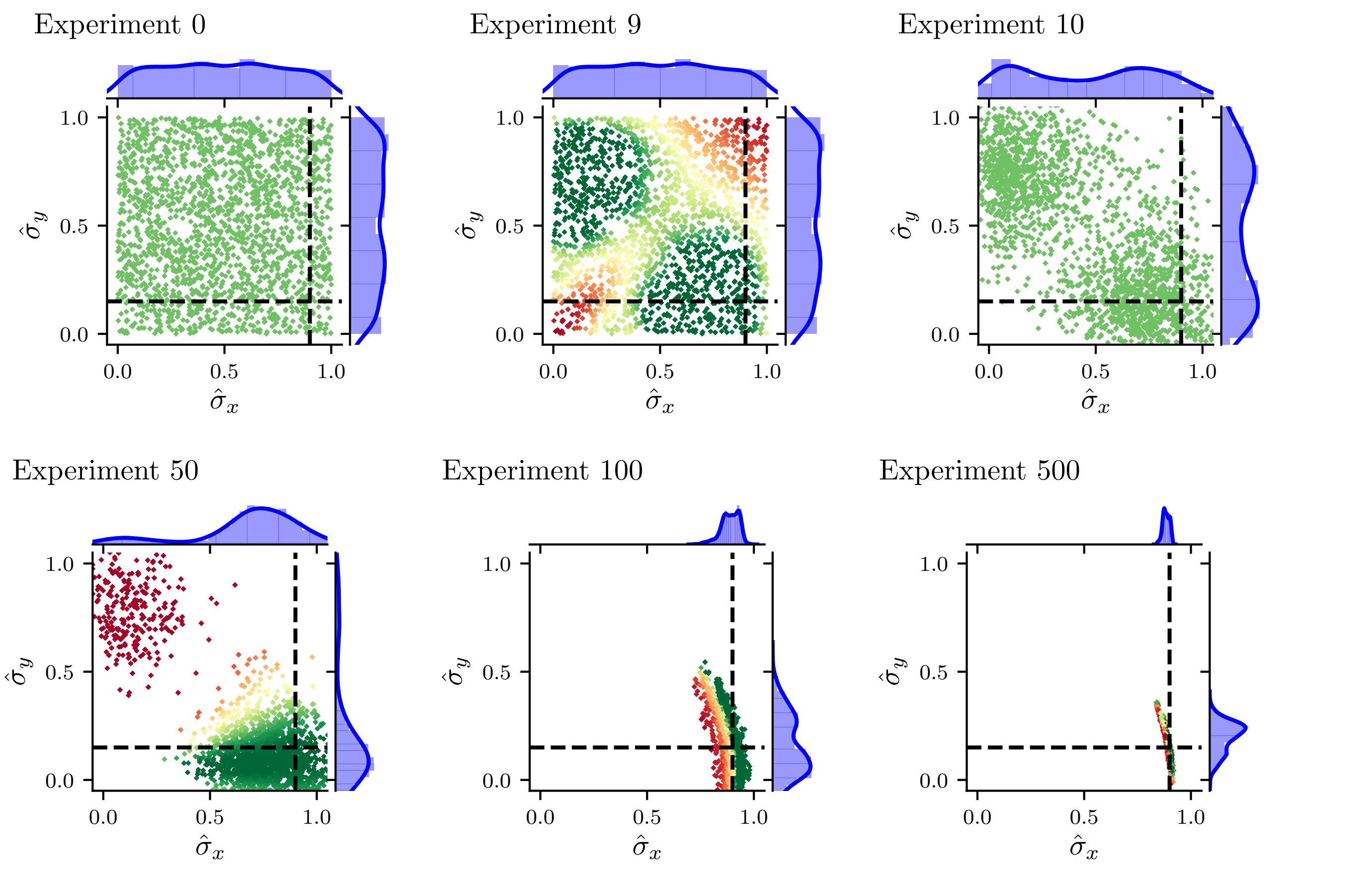
which can be simulated clasically or using a quantum simulator (see Section 1.3). It is necessary first to know the datum d (either 0 or 1) which was projected by Q under experimental conditions. Therefore we first perform the experiment e on Q (preparing the state $|\psi\rangle$ evolving for t and projecting again onto $\langle\psi|$) to retrieve the datum d. d is then used for the calculation of the likelihood for each particle sampled from $Pr(\vec{\alpha})$. Each particle's probability can be updated by Eq. (1.3), allowing us to redraw the entire probability distribution. We can hence compute a *posterior* probability distribution by performing this routine on a set of N_p particles: we hypothesise N_p parameterisations $\vec{\alpha}_i$ sampled from $\Pr(\vec{\alpha})$, and update their $\Pr(\vec{\alpha}_i)$ in proportion to their likelihood. In effect, hypotheses (particles) which are found to be highly likely are given increased credence, while those with low likelihood have their credence decreased.

1.2 SEQUENTIAL MONTE CARLO

In practice, QHL samples from and updates $\Pr(\vec{\alpha})$ via sequential monte carlo (SMC). SMC samples the N_p particles from $\Pr(\vec{\alpha})$, and assigns each particle a weight, $w_0 = 1/N_p$. Each particle corresponds to a unique position in the parameters' space, i.e. $\vec{\alpha}_p$. Following the calculation of the likelihood, $\Pr(d|\vec{\alpha}_p;e)$, the weight of particle p is updated from its initial value of w_p^{old} by Eq. (1.5).

$$w_p^{\text{new}} = \frac{\Pr(d|\vec{\alpha}_p;e) \times w_p^{\text{old}}}{\sum\limits_p w_p \Pr(\vec{\alpha}_p|d;e)} \tag{1.5}$$

In this way, strong particles – with high $\Pr(d|\vec{\alpha}_p;e)$ – have their weight increased, while weak particles (low $\Pr(d|\vec{\alpha}_p;e)$) have their weights decreased, and the sum of weights remains normalised. Within a single experiment, the weights of all N_p particles are updated: we *simulataneously* update sampled particles' weights as well as $\Pr(\vec{\alpha})$. The procedure of updating particles' weights iterates for the subsequent experiment, using the *same* particles: we do *not* redraw N_p particles for every experiment. Eventually, the weights of most particles fall below a threshold, r_t, meaning that only that fraction of particles have reasonable likelihood of being $\vec{\alpha}_0$. At this stage, SMC *resamples*, i.e. selects new particles, according to the updated $\Pr(\vec{\alpha})$[2]. Then, the new particles are in the range of parameters which is known to be more likely, while particles in the region of low-weight are effectively discarded. Usually, we set $r_t = 0.5$, although



very

Figure 1.1: Quantum Hamiltonian learning (QHL) via sequential monte carlo (SMC). The studied model has two terms, $\{\hat{\sigma}_x, \hat{\sigma}_y\}$ with true parameters $\alpha_x = 0.9, \alpha_y = 0.15$ (dahsed lines), with resources $N_e = 500, N_p = 2000$ for training the model. Crosses represent particles, while the distribution $\Pr(\alpha_p)$ for each parameter can be seen along the top and right-hand-sides of each subplot. Both parameters are assigned a uniform probability distribution $\mathcal{U}(0,1)$, representing our prior knowledge of the system. **(a),** sequential monte carlo (SMC) samples N_p particles from the initial joint probability distribution, with particles uniformly spread across the unit square, each assigned the starting *weight* w_0. At each experiment e, each of these particles' likelihood is computed according to Eq. (1.3) and its weight is updated by Eq. (1.5). **(b),** after 9 experiments, the weights of the sampled particles are sufficiently informative that we know we can discard some particles while most likely retaining the true parameters. **(c),** SMC resamples according the current $\Pr(\vec{\alpha})$, i.e. having accounted for the experiments and likelihoods observed to date, a new batch of N_p particles are drawn, and each reassigned weight w_0, irrespective of their weight prior to resampling. **(d, e),** Afer further experiments and resamplings, SMC narrows $\Pr(\vec{\alpha})$ to a region around the true parameters. **(f),** The final *posterior* distribution consists of two narrow distributions centred on α_x and α_y. By taking the mean of the posterior distribution, we approximate the parameters of interest as $\vec{\alpha}'$.

this hyperparameter can have a large impact on the rate of learning, so can be optimised in particular circumstances, see Fig. 1.2.

This procedure is easiest understood through the example presented in Fig. 1.1, where a two-parameter Hamiltonian is learned starting from a uniform distribution. $N_p = 2000$ particles are used to propose hypotheses distributed evenly throughout the parameter space, each of which are subject to weight updates as outined above. In this example, after 9 experiments the particles around the diagonal ($x = y$) are deemed unlikely, while clusters form in the opposite corners where the algorithm finds the hypotheses credible. Before the tenth experiment, the algorithm resamples, i.e. reassigns weights based on the present $\Pr(\vec{\alpha})$. The algorithm iteratively reassigns weight to particles based on their likelihoods, redraws $\Pr(\vec{\alpha})$ and resamples. We show the state of the particles after 50, 100 and 500 experiments, with the overall result of a highly peaked parameter distribution, whose centre is near the target parameters.

1.3 LIKELIHOOD

The fundamental step within QHL is the calculation of likelihood, which enables updates of the probability distribution in Eq. (1.3). The key to the learning algorithm is that likelihood can be retrieved from the Born rule, which captures how likely a given a quantum system is to be measured in an eigenstate. When we have retrieved a datum, d, from Q, we can compute the probability that Q would be measured in the corresponding eigenstate $|d\rangle$ – this probability serves as the likelihood, and is given by Eq. (1.4).

In some cases, it is feasible to derive the closed form of the likelihood, for example as a simple expression in terms of the Hamiltonian parameters, which we will exemplify in Section 1.3.2. Closed form likelihoods allow for rapidly testing hypothetical parameters for comparison against the observed data, so QHL can feasibly be run with high N_e, N_p. In general, however, it is not possible to derive the closed form of the likelihood, and instead the likelihood must be computed through Eq. (1.4), which can be done either on a classical or quantum simulator. The case where the likelihood is computed on a quantum simulator is referred to as quantum likelihood estimation (QLE) [3, 5], and can leverage any algorithm for the calculation of Hamiltonian dynamics to achieve *quantum speedup* [7, 8].

In this thesis, we do not implement the presented algorithms on quantum hardware, instead investigating their performance using idealised classical simulations, i.e. classical likelihood estimation (CLE). The reliance on classical resources demands that Eq. (1.4) be computed explicitly, notably involving the matrix exponential $e^{-i\hat{H}(\vec{\alpha}_p)t}$. Since the Hamiltonian matrix scales with the size of the simulated system, running QHL for an n-qubit systems requires exponentiation of its $2^n \times 2^n$ Hamiltonian matrix, in order to compute the exact likelihoods required for learning. This overhead restricts the applicability of CLE: $n = 11$-qubit systems'

2 Particles are *resampled* according to a resampling algorithm. Throughout this thesis, we always use the Liu-West resampling algorithm [6].

Hamiltonians exhaust the memory capacity of most conventional classical computers. In practice, QHL is limited by the computation of the total $N_e N_p$ matrix exponentials required for training: we will only entertain systems which can be represented by Hamiltonians of up to $n = 8$ qubits. In principle, larger systems could be condensed for simulation on available classical resources, or those resources used more efficiently [9], but the remit of this thesis can be fulfilled with demonstrations in the domain $n \leq 8$ qubits, so we do not endeavour to find the most effective classical strategies.

Adopting the notation used by QInfer [10], upon which our software builds, the expectation value for a the unitary operator is given by

$$\Pr(0) = | \langle\psi| \, e^{-i\hat{H}_p t} \, |\psi\rangle \, |^2 = l(d = 0|\hat{H}_p; e). \tag{1.6}$$

In Eq. (1.6), the input basis is assigned the measurement label $d = 0$, and this $\Pr(0)$ is the probability of measuring $d = 0$, i.e. measuring the same state as was prepared as input. We assume a binary outcome model[3], i.e. that the system is measured either in $|\psi\rangle$ (labelled $d = 0$), or it is not ($|\psi_\perp\rangle \, , d = 1$); the likelihood for the latter case is

$$\Pr(1) = l(d = 1|\hat{H}_p; e) = \sum_{\{|\psi_\perp\rangle\}} | \langle\psi_\perp| \, e^{-i\hat{H}_p t} \, |\psi\rangle \, |^2 = 1 - \Pr(0). \tag{1.7}$$

Usually we will refer to the case where Q is projeced onto the input state $|\psi\rangle$, so the terms *likelihood*, *expectation value* and $\Pr(0)$ are synonymous, unless otherwise stated.

1.3.1 *Interactive Quantum likelihood Estimation*

A fundamental result in quantum mechanics (QM) – the Loschmidt echo (LE) – shows that marginally differing Hamiltonians produce exponentially diverging evolutions, undermining the basis of QLE, i.e. that the likelihood function can inform Bayesian updates to a paramater distribution. The LE concerns the result when Q is prepared in some initial state $|\psi\rangle$, evolved forward in time by some $\hat{H}_+$, then evolved *backwards*[4] in time by $\hat{H}_-$, and projected back onto $|\psi\rangle$. The LE – or the *fidelity* – is given by

$$M(t) = \left| \langle\psi| e^{+i\hat{H}_- t} e^{-i\hat{H}_+ t} |\psi\rangle \right|^2 . \tag{1.8}$$

3 In principle the output does not have to be binary, so we sum over the general set $\{|\psi_\perp\rangle\}$ of eigenstates orthogonal to $|\psi\rangle$ in Eq. (1.7).

4 Equivalently and in practice, evolved forward in time for $-\hat{H}_-$.

$M(t)$ is dictated by the *similarity* between the two Hamiltonians. If $\hat{H}_+ = \hat{H}_-$, then $M(t) = 1$, while $\|\hat{H}_+ - \hat{H}_-\|_2 > 0$ yields $M(t) < 1$, indicating disagreement between the two Hamiltonians. The fidelity is characterised by a number of distinct regions, depending on the evolution time, t:

$$M(t) \sim \begin{cases} 1 - \mathcal{O}(t^2), & t \leq t_c \\ e^{-\mathcal{O}(t)}, & t_c \leq t \leq t_s \\ 1/\|\hat{H}\|, & t \geq t_s \end{cases} \tag{1.9}$$

where $\|\hat{H}\|$ is the dimension of the Hamiltonians, and t_c, t_s are bounds on the evolution time marking the transition between the *parabolic decay*, *asymptotic decay* and *saturation* of the echo [11]. t_c and t_s generally depend on the similarity between $\hat{H}_+$ and $\hat{H}_-$: intuitively, as $\|\hat{H}_+ - \hat{H}_-\|_2$ decreases, the echo does not saturate until higher evolution times.

Recall that the Bayesian updates to the parameter distribution relies on good hypotheses receiving likelihood $l_e \sim 1$, and weak hypothesis receiving $l_e \sim 0$. The LE tells us that there is a small range of evolution times ($t \lesssim t_c$) for which even good particles may expect $l_e \sim 1$. We can exploit this effect, however: by designing experiments with $t \sim t_c$, the likelihood is extremely sensitive to the parameterisation, in that only particles close to the precise parameters will give a high likelihood in this regime. This is the basis of the particle guess heuristic, described in Section 1.6.1.

We can relate the LE to the likelihood, Eq. (1.4), by supposing $\hat{H}_- = \hat{\mathbb{1}}$. It is inescapable that the likelihoods are exponentially small if the evolution times are not short; experimentally, exponentially small expectation values demand an exponential number of measurements to approximate accurately. Furthermore, short-time experiments are known to be uninformative [2, 12]. Together, these problems render QLE unscalable. We overcome these inherent problems by using a modification of QLE, interactive quantum likelihood estimation (IQLE), the key to which is invoking a likelihood function other than Eq. (1.4).

In effect, the LE guarantees that, for most t, if $\hat{H}_- \not\approx \hat{H}_+$, then $M(t) \ll 1$, while $\hat{H}_- \approx \hat{H}_+$ gives $M(t) \approx 1$. This can be exploited for learning: by taking $\hat{H}_+$ as either $\hat{H}_0$ (the true system) or $\hat{H}(\vec{\alpha})$ (particle/hypothesis), and sampling $\hat{H}_-$ from $\Pr(\vec{\alpha})$, we can adopt Eq. (1.8) as the likelihood function. The likelihood that they are both measured in the same eigenstate is still a function of the overlap between the Then, both $\hat{H}_0$ and $\hat{H}(\vec{\alpha})$ have been evolved for arbitrary t, and unevolved by a common unitary, $e^{i\hat{H}_+ t}$. hypothesis and the true parameters, but here the informative difference between them is not drowned out by the chaotic effects captured by the LE, as it had been in QLE.

Importantly, IQLE can only be used where we can *reliably* reverse the evolution for the system under study. In order that the reverse evolution is relibable, it must be performed on a trusted simulator, restricting IQLE to cases where a coherent quantum channel exists between the target system and a trusted simulator. This automatically excludes any open quantum systems, as well as most realistic experimental setups, although such channels can be achieved [13]. The remaining application for IQLE, and correspondingly QHL, is in the characterisation of untrusted quantum simulators, which can realise such coherent channels [5].

1.3.2 *Analytical likelihood*

For some Hamiltonians, we can derive an analytical likelihood function to describe their dynamics [14, 15]. For instance, the Hamiltonian for an oscillating electron spin in a nitrogen-vacancy centre is given by

$$\hat{H}(\omega) = \frac{\omega}{2}\hat{\sigma}_z, \tag{1.10}$$

where ω is the Rabi frequency of the spin. Then, recalling that $\hat{\sigma}_z\hat{\sigma}_z = \hat{\mathbb{1}}$, so $\hat{\sigma}_z^{2k} = \hat{\mathbb{1}}$ and $\hat{\sigma}_z^{2k+1} = \hat{\sigma}_z$, using MacLaurin expansion, the unitary evolution of Eq. (1.10) is given by

$$\begin{aligned} U = e^{-i\hat{H}(\omega)t} = e^{-i\frac{\omega t}{2}\hat{\sigma}_z} &= \cos\left(\frac{\omega t\hat{\sigma}_z}{2}\right) - i\sin\left(\frac{\omega t\hat{\sigma}_z}{2}\right) \\ &= \left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k)!}\left(\frac{\omega t}{2}\right)^{2k}\hat{\sigma}_z^{2k}\right) - i\left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)!}\left(\frac{\omega t}{2}\right)^{2k+1}\hat{\sigma}_z^{2k+1}\right) \\ &= \left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k)!}\left(\frac{\omega t}{2}\right)^{2k}\right)\hat{\mathbb{1}} - i\left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)!}\left(\frac{\omega t}{2}\right)^{2k+1}\right)\hat{\sigma}_z \\ &= \cos\left(\frac{\omega t}{2}\right)\hat{\mathbb{1}} - i\sin\left(\frac{\omega t}{2}\right)\hat{\sigma}_z \end{aligned} \tag{1.11}$$

Then, evolving a probe $|\psi_0\rangle$ and projecting onto a state $|\psi_1\rangle$ gives

$$\langle\psi_1|\,U\,|\psi_0\rangle = \cos\left(\frac{\omega t}{2}\right)\langle\psi_1|\psi_0\rangle - i\sin\left(\frac{\omega t}{2}\right)\langle\psi_1|\hat{\sigma}_z|\psi_0\rangle\,. \tag{1.12}$$

By initialising and projecting into the same state, say $|\psi_0\rangle = |\psi_1\rangle = |+\rangle$, and reaclling $\hat{\sigma}_z\,|+\rangle = |-\rangle$, we have

$$\begin{aligned} \langle\psi_1|\psi_0\rangle &= \langle+|+\rangle = 1 \\ \langle\psi_1|\hat{\sigma}_z|\psi_0\rangle &= \langle+|-\rangle = 0 \\ \implies \langle\psi_1|U|\psi_0\rangle &= \cos\left(\frac{\omega t}{2}\right), \end{aligned} \tag{1.13}$$

Then, if the system measures in $|+\rangle$, we set the datum $d = 0$, otherwise $d = 1$. From Born's rule, and in analogy with Eq. (1.4), we can formulate the likelihood function, where the hypothesis is the single parameter ω, and the sole experimental control is t,

$$\Pr(d = 0|\omega;t) = |\langle\psi_1|U|\psi_0\rangle|^2 = \cos^2\left(\frac{\omega t}{2}\right) \tag{1.14a}$$

$$\Pr(d = 1|\omega;t) = 1 - \cos^2\left(\frac{\omega t}{2}\right) = \sin^2\left(\frac{\omega t}{2}\right) \tag{1.14b}$$

This analytical likelihood will underly the simulations used in the following introductions, except where explicitly mentioned.

1.4 TOTAL LOG TOTAL LIKELIHOOD

We have already used the concept of likelihood to update our parameter distribution during SMC; we can consolidate the likelihoods of all particles with respect to a single datum, d, from a single experiment e, in the *total likelihood*,

$$l_e = \sum_{p \in \{p\}} \Pr(d|\vec{\alpha}_p; e) \times w_p^{\text{old}}, \tag{1.15}$$

where w_p^{old} are the particle *weights* for the particle with parameterisation $\vec{\alpha}_p$. For each experiment, we use total likelihood as a measure of how well the distribution performed, i.e. we care about how well all particles, $\{p\}$, perform as a collective, representative of how well $\Pr(\vec{\alpha})$ approximates the system, equivalent to the noramlisation factor in Eq. (1.5), [16].

l_e are strictly positive, and because the natural logarithm is a monotonically increasing function, we can equivalently work with the log total likelihood (LTL), since $ln(l_a) > ln(l_b) \iff l_a > l_b$. LTL are also beneficial in simplifying calculations, and are less susceptible to system underflow, i.e. very small values of l will exhaust floating point precision, but $ln(l)$ will not.

Note, we know that

$$\begin{aligned} w_p^0 = \frac{1}{N_p} &\implies \sum_p^{N_p} w_p^0 = 1; \\ eqn : likelihood_deriv \quad \Pr(d|\vec{\alpha}_p; e) \leq 1 &\implies \Pr(d|\vec{\alpha}_p; e) \times w_p^{\text{old}} \leq w_p^{\text{old}} \\ &\implies \sum_{\{p\}} \Pr(d|\vec{\alpha}_p; e) \times w_p^{\text{old}} \leq \sum_{\{p\}} w_p^{\text{old}} \leq \sum_p^{N_p} w_p^0; \\ &\implies l_e \leq 1. \end{aligned} \tag{1.16}$$

?? essentially says that a good batch of particles, where on average particles perform well, will mean that most w_i are high, so $l_e \approx 1$. Conversely, a poor batch of particles will have low average w_i, so $l_e \approx 0$.

In order to assess the quality of a *model*, $\hat{H}_i$, we can consider the performance of a set of particles throughout a set of experiments $\mathcal{E}$, through its total log total likelihood (TLTL),

$$\mathcal{L}_i = \sum_{e \in \mathcal{E}} ln(l_e). \tag{1.17}$$

The set of experiments on which $\mathcal{L}_i$ is computed, $\mathcal{E}$, as well as the particles whose sum constitute each l_e, can be the same experiments on which $\hat{H}_i$ is trained, $\mathcal{E}_i$, but in general need not be. That is, $\hat{H}_i$ can be evaluated by considering different experiments than those on which it was trained. For example, $\hat{H}_i$ can be trained with $\mathcal{E}_i$ to optimise $\vec{\alpha}_i'$, and thereafter be evaluated using a different set of experiments $\mathcal{E}_v$, such that $\mathcal{L}_i$ is computed using particles sampled from

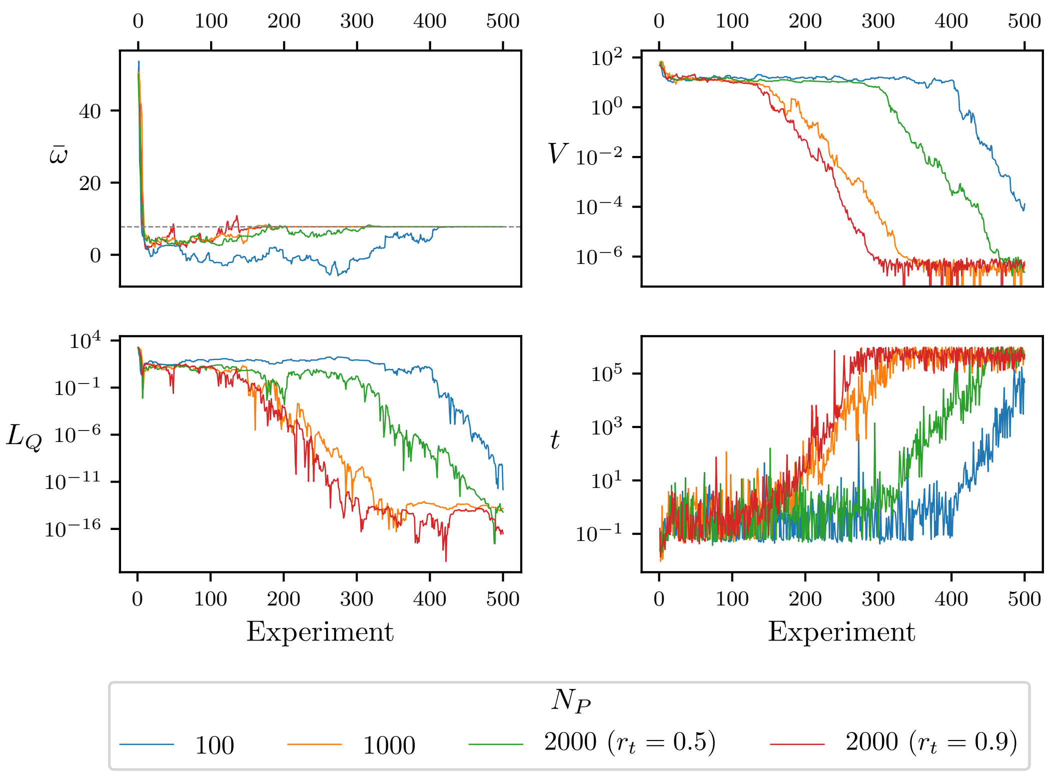


Figure 1.2: Parameter learning for the analyitcal likelihood Eq. (1.14) for varying numbers of particles N_p, for $N_e = 500$. For $N_p = 2000$, we show the resampler threshold set to $r = 0.5$ and $r = 0.9$. (a) the parameter estimate, i.e. $\bar{\omega}$, the mean of the posterior distribution after each experiment, approaching $\omega_0 = 7.75$ (dashed line), where the prior is centred on $\omega = 50 \pm 25$. Decrease in (b) volume, V, (c) quadratic loss, L_Q, and (d) evolution time, t, are shown against experiment number. Implementation details are listed in Table A.1

the distribution after optimising $\vec{\alpha}$, $\Pr(\vec{\alpha}_i')$, and may use a different number of particles than the training phase.

Perfect agreement between the model and the system would result in $l_e = 1 \Rightarrow ln(l_e) = 0$, as opposed to imperfect agreemenet where $l_e < 1 \Rightarrow ln(l_e) < 0$. In all cases Eq. (1.17) is negative, and across a series of experiments, strong agreement gives low $|\mathcal{L}_i|$, whereas weak agreement gives large $|\mathcal{L}_i|$.

1.5 PARAMETER ESTIMATION

QHL is a parameter estimation algorithm, so here we introduce some methods to evaluate its performance, which we can reference in later sections of this thesis. The most obvious meausre of the progression of parameter estimation is the error between the true parameterisation, $\vec{\alpha}_0$, and the approximation $\vec{\alpha}_p = \text{mean}\left(\Pr(\vec{\alpha})\right)$, which can be captured by a large family of loss functions. Among others, we use the quadratic loss (QL), which captures this error through the sum of the square difference between each parameters' true and estimated values symetrically. We can record the QL at each experiment of our training regime and hence track its over- or under-esimtation. The QL is given by

$$L_Q(\vec{\alpha}) = \|\vec{\alpha}_0 - \vec{\alpha}\|^2 \tag{1.18}$$

where $\vec{\alpha}_0$ is the true parameterisation and $\vec{\alpha}$ a hypothesis distribution.

1.5.1 *Volume*

We also care about the range of parameters supported by $\Pr(\vec{\alpha})$ at each experiment: the volume of the particle distribution can be seen as a proxy for our certainty that the approximation mean $(\Pr(\vec{\alpha}))$ is accurate. For example, for a single parameter ω, our best knowledge of the parameter is mean $(\Pr(\omega))$, and our belief in that approximation is the standard deviation of $\Pr(\omega)$; we can think of volume as an n-dimensional generalisation of this intuition [10, 17].

In general, a confidence region, defined by its confidence level κ, is drawn by grouping particles of high particle density (HPD), $\mathcal{P}$, such that $\sum\limits_{p\in\mathcal{P}} w_p \geq \kappa$. We use the concept of *minimum volume enclosing ellipsoid* to capture the confidence region [17], calculated as in [18], which are characterised by their covariance matrix, Σ, which allows us to calculate the volume,

$$V(\Sigma) = \frac{\pi^{|\vec{\alpha}|/2}}{\Gamma(1+\frac{|\vec{\alpha}|}{2})} \det\left(\Sigma^{-\frac{1}{2}}\right), \tag{1.19}$$

where Γ is the Gamma function, and $|\vec{\alpha}|$ is the cardinality of the parameterisation. This quantity allows us to meaningfully compare distributions of different dimension, but we must be cautious of drawing strong comparisons between models based on their volume alone, for instance because they may have started from vastly different prior distributions.

Within SMC, we assume the credible region is simply the posterior distribution, such that we can take $\Sigma = \text{cov}(\Pr(\vec{\alpha}))$ after each experiment, and hence track the uncertainty in our parameters across the training experiments [1]. We use volume as a measure of the learning procedure's progress: slowly decreasing or static volume indicates poor learning, possibly highlighting poor experiment design, while decreasing volume indicates that the parameters' estimation is improving. When the volume has converged, e.g. the red model in Fig. 1.2, the learning has saturated and there is little benefit to running further experiments.

1.6 EXPERIMENT DESIGN HEURISTIC

A key consideration in QHL is the choice of experimental controls implemented in attempt to learn from the system. The experimental controls required are dictated by the choice of likelihood function used within SMC, though typically there are two primary controls we will focus on: the evolution time, t, and the *probe* state evolved, $|\psi\rangle$. The design of experiments is handled by an experiment design heuristic (EDH), whose structure can be altered to suit the user's needs, with respect to the individual target system. Usually, the EDH attempts to exploit the information available, adaptively accounting for some aspects of the inference process performed already. In some cases, however, there may be justification to employ a non-adaptive schedule, for instance to force QHL to train upon a full set of experimental data rather than a subset, as an adaptive method may advise. We can categorise each EDH as either *online* of *offline*, depending on whether it accounts for the current state of the inference procedure, i.e. the posterior. The EDH is modular and can be replaced by any method that returns a valid set of experimental controls, so we can consider numerous approaches, for instance those described in [19, 20].

1.6.1 *Particle Guess Heuristic*

The default EDH is the particle guess heuristic (PGH) [3], an online method which attempts to design the optimal evolution time based on the posterior at each experiment. Note PGH does not specify the probe, so is coupled with a probe selection routine to comprise a complete EDH.

The principle of PGH is that the uncertainty of the posterior limits how well the Hamiltonian is currently approximated, and therefore limits the evolution time for which the posterior can be expected to reasonably mimic $\hat{H}_0$[5]. For example, consider Eq. (1.10) with a single parameter with $\omega_0 = 10$, and current $\text{mean}\left(\Pr(\omega)\right) = 9, \text{std}\left(\Pr(\omega)\right) = 2$: we can expect that the approximation $\omega' = \text{mean}\left(\Pr(\omega)\right)$ is valid up to $t_{max} \approx 1/\text{std}(\Pr(\omega))$. It is sensible, then, to use $t \sim t_{max}$ for two reasons: (i) smaller times are already well explained by the posterior, so offer little opportunity to learn; (ii) t_{max} is at or near the threshold which particles sampled from the posterior can comfortably explain, so it will expose the relative difference in likelihood between the posterior's better and worse particles, providing a capacity to learn. Informally, as the uncertainty in the posterior shrinks, PGH selects larger times to ensure the training is based on informative eperiments while simultaneously increasing certainty about the paraemters. In the one-dimensional case, this logic can be used to find an optimal time heuristic, where experiment k is assigned $t_k = 1.26/\text{std}(\Pr(\omega))$ [15].

For a general multidimensional parameterisation, rather than directly using the inverse of the standard deviation of $\Pr\left(\vec{\alpha}\right)$, which relies on the expensive calculation of the covarinace matrix, PGH uses a proxy whereby two particles are sampled from $\Pr\left(\vec{\alpha}\right)$. The experimental evolution time for experiment k is then given by

$$t_k = \frac{1}{\|\vec{\alpha}_i - \vec{\alpha}_j\|}, \tag{1.20}$$

5 The reasoning behind limiting the evolution time according to the posterior distribution is rooted in the effect of the Loschmidt echo, described in Section 1.3.1.

where $\vec{\alpha}_i, \vec{\alpha}_j$ are distinct particles sampled from $\mathcal{P}$ where $\mathcal{P}$ is the set of particles under consideration by SMC after experiment $k-1$, which had been recently sampled from $\Pr(\vec{\alpha})$.

1.6.2 *Alternative experiment design heuristics*

The EDH can be specified to the requirements of the target system; we test four examples of customised EDHs against four target Hamiltonians. Here the EDH must only design the evolution time for the experiment, with probe design discussed in the next section. The heuristics tested are:

- Random$(0, t_{max})$: Randomly chosen time up to some arbitrary maximum, we set $t_{max} = 1000$ (arbitrary units). This approach is clearly subobtimal, since it does not account whatsoever for the knowledge of the training so far, and demands the user choose a suitable t_{max}, which can not gauranteed to be meaningful.
- t list: forcing the training to consider a set of times decided in advance. For instance, when only a small set of experimental measurements are available, it is sensible to train on all of them, perhaps repeatedly. We test uniformly spaced times between 0 and t_{max}, and cycle through the list twice, aiming first to broadly learn the region of highest likelihood for all times, and then to refine the approximation. Again this EDH fails to account for the performance of the trainer so far, so may use times either far above or below the ability of the parameterisation.
- $(9/8)^k$: An early attempt to match the expected exponential decrease in volume from the training, was to set $t_k = (9/8)^k$ [1]. Note we increment k after 10 experiments in the training regime, rather than after each experiment, which would result in extremely high times which flood central processing unit (CPU) memory.
- PGH: as in Section 1.6.1.

We demonstrate the influence of the EDH on the training procedure by testing models of various complexity amd dimension in Fig. 1.3[6]. In particular, we first test a simple 1-qubit model, Eq. (1.21a); followed by more complicated 1-qubit model, Eq. (1.21b); as well as randomly generated 5-qubit Ising, Eq. (1.21c), and 4-qubit Heisenberg models, Eq. (1.21d). Each $\hat{H}_i$ have randomly chosen parameters implicitly assigned to each term.

$$\hat{H}_1 = \hat{\sigma}_1^z \tag{1.21a}$$

$$\hat{H}_2 = \hat{\sigma}_1^x + \hat{\sigma}_1^y + \hat{\sigma}_1^z \tag{1.21b}$$

$$\hat{H}_3 = \hat{\sigma}_1^z\hat{\sigma}_3^z + \hat{\sigma}_1^z\hat{\sigma}_4^z + \hat{\sigma}_1^z\hat{\sigma}_5^z + \hat{\sigma}_2^z\hat{\sigma}_4^z + \hat{\sigma}_2^z\hat{\sigma}_5^z + \hat{\sigma}_3^z + \hat{\sigma}_4^z + \hat{\sigma}_3^z + \hat{\sigma}_5^z \tag{1.21c}$$

6 Note the models designed here are not intended to represent physically meaningful situations, but merely to serve as examples of simulatable Hamiltonians.

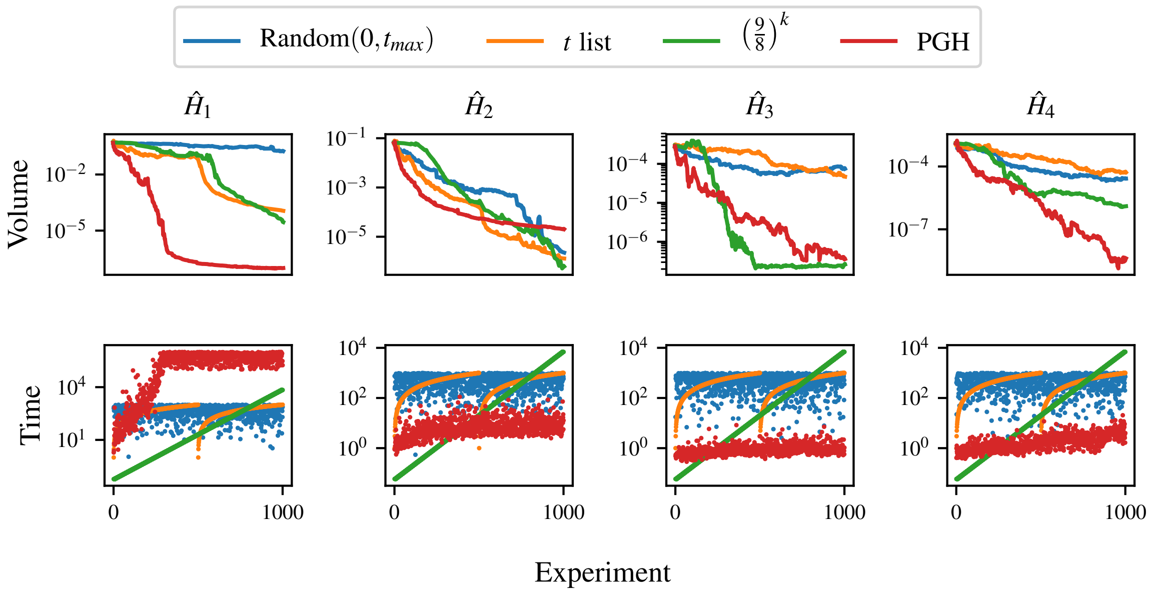


Figure 1.3: The volume and evolution times of various models when trained through QHL using different EDHs. We show models of various complexity and dimension, each trained using four heuristics, outlined in the main text. Implementation details are listed in Table A.1

$$\hat{H}_4 = \hat{\sigma}_1^z\hat{\sigma}_2^z + \hat{\sigma}_1^z\hat{\sigma}_3^z + \hat{\sigma}_2^x\hat{\sigma}_3^x + \hat{\sigma}_2^z\hat{\sigma}_3^z + \hat{\sigma}_2^x\hat{\sigma}_4^x + \hat{\sigma}_3^z\hat{\sigma}_4^z \tag{1.21d}$$

We show the performance of each of the listed EDHs in Fig. 1.3. The general trend reveals that, although some individual models benefit from bespoke EDHs, the PGH is generically applicable and usually facilitates a reasonable level of training, without providing advantage to any model. We will have cause to use alternative EDHs in paricular circumstances, but we adopt PGH as the default EDH throughout this thesis, unless otherwise stated.

1.7 PROBE SELECTION

A final consideration about training experiments within QHL is the choice of input probe state, $|\psi\rangle$, which is evolved in the course of finding the likelihood used during the Bayesian update. We can consider the choice of probe as an output of the EDH, although previous work has usually not considered optimising the probe, instead usually setting $|\psi\rangle = |+\rangle^{\otimes n}$ for n qubits [5, 15]. In principle it is possible for the EDH to design a new probe at each experiment, although a more straightforward approach is to compose a set of probes offline, $\Psi = \{|\psi\rangle\}$, of size $N_\psi = |\Psi|$. Then, a probe is chosen at each experiment from Ψ, allowing for the same $|\psi\rangle$ to be used for multiple experiments within the training, e.g. by iterating over Ψ. Ψ can be generated with

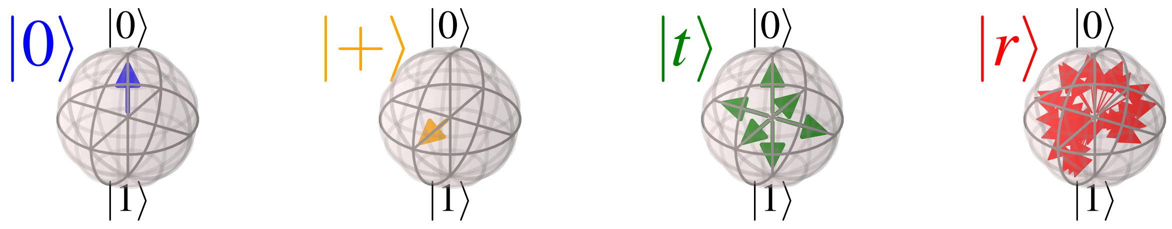


Figure 1.4: 1-qubit probes used for tests in Fig. 1.5.

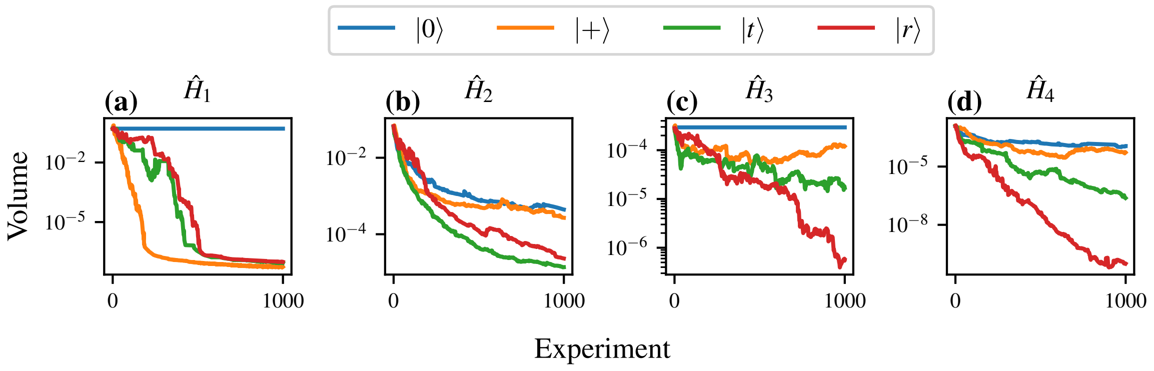


Figure 1.5: The volume of various models when trained through QHL using different initial probe sets. We show models of various complexity and dimension, each trained using random probes, $|r\rangle$, tomographic basis set probes, $|t\rangle$, as well as $|0\rangle$ and $|+\rangle$ probes. In each case the probes are generated for arbitrary numbers of qubits; for $|0\rangle, |1\rangle$, the number of probes generated is $N_\psi = 1$, and for $|t\rangle, |r\rangle$, $N_\psi = 40$. Implementation details are listed in Table A.1

respect to the individual learning problem as we will examine later, but it is usually sufficient to use generic strategies which should work for all models; some straightforward examples are

i. $|0\rangle : \ \Psi = \{|0\rangle^{\otimes n}\}, \ \ N_\psi = 1;$

ii. $|+\rangle : \ \Psi = \{|+\rangle^{\otimes n}\}, \ \ N_\psi = 1;$

iii. $|t\rangle$: Ψ is a random subset of probes generated by combining tomographic basis states, $N_\psi = 40$;

iv. $|r\rangle$: $|\psi\rangle$ are random, separable probes, $N_\psi = 40$.

We show the 1-qubit probes within Ψ under each of these strategies on the Bloch sphere in Fig. 1.4.

Recalling the set of models from Eq. (1.21), we test each of these probe construction strategies in Fig. 1.5. We can draw a number of useful observations from these simple tests:

- Training on an eigenstate – as in the case for $\hat{H}_1$ and $\hat{H}_3$ using $|0\rangle$ – yields no information gain. This is because all particles give likelihoods $l = 1$, so no weight update can occur, meaning the parameter distribution does not change when presented new evidence.
- Training on an even superposition of the model's eigenstates – e.g. $|+\rangle$ for $\hat{H}_1$ – is maximally informative: any deviations from the true parameterisation are registered most dramatically in this basis, providing the optimal training probe for this case.
- These observations are reinforced by Fig. 1.5**c**, where a 5-qubit Ising model also fails to learn from one of its eigenstates, $|0\rangle^{\otimes 5}$. Of note, however, is that $|+\rangle^{\otimes 5}$ is not the strongest probe here: the much larger Hilbert space here can not be scanned sufficiently using a single probe; using a larger number of probes is more effective, even if those are randomly chosen.
- In general the tomographic and random probe sets perform reliably, even for complex models.

It is an open challenge to identify the optimal probe for training any given model; the design of informative probes could be built into the EDH in principle, e.g. a set of probes could be generated of even superpositions of the candidate's eigenstates. However, for model comparison purposes in general, it is helpful to have a universal set of probes, Ψ, upon which all models are trained. The use of Ψ minimises systematic bias towards particlar models, which might arise from probes which server as favourable bases for a subset of models, for example $|+\rangle$ in Fig. 1.5**a**. Careful consideration should be given to N_ψ in the choice of the probe generator, since it is important to ensure probes robustly test the parameterisation across the entire Hilbert space. It is also necessary that SMC has sufficient opportunity to learn within a given subspace before moving to the next, so that slight deviations in $\Pr(\vec{\alpha})$ due to a single probe are not immediately reversed because a distant probe is immediately invoked. We can mitigate this concern by instructing the EDH to repeatedly select a probe from Ψ for a batch of successive experiments, before moving to the next available probe. For the remainder of this thesis, unless otherwise stated, we adopt the random probe generator as the defualt mechanism for selecting probes, iterating between probes after batches of 5 experiments.

APPENDIX

A FIGURE REPRODUCTION

Most of the figures presented in the main text are generated directly by the Quantum Model Learning Agent (QMLA) framework. Here we list the implementation details of each figure so they may be reproduced by ensuring the configuration in Table A.1 are set in the launch script. The default behaviour of QMLA is to generate a results folder uniquely identified by the date and time the run was launched, e.g. results can be found at the *results directory* qmla/Launch/Jan_01/12_34. Given the large number of plots available, ranging from high-level run perspective down to the training of individual models, we introduce a plot_level $\in \{1, ..., 6\}$ for each run of QMLA: higher plot_level informs QMLA to generate more plots.

Within the results directory, the outcome of the run's instances are stored, with analysis plots broadly grouped as

- evaluation: plots of probes and times used as the evaluation dataset.
- single_instance_plots: outcomes of an individual QMLA instance, grouped by the instance ID. Includes results of training of individual models (in model_training), as well as subdirectories for anlaysis at the branch level (in branches) and comparisons.
- combined_datasets: pandas dataframes containing most of the data used during analysis of the run. Note that data on the individual model/instance level may be discarded so some minor analyses can not be performed offline.
- exploration_strategy_plots plots specifically required by the exploration strategy (ES) at the run level.
- champion_models: analysis of the models deemed champions by at least one instance in the run, e.g. average parameter estimation for a model which wins multiple instances.
- performance: evaluation of the QMLA run, e.g. the win rate of each model and the number of times each term is found in champion models.
- meta analysis of the algorithm' implementation, e.g. timing of jobs on each process in a cluster; generally users need not be concerned with these.

In order to produce the results presented in this thesis, the configurations listed in Table A.1 were input to the launch script. The launch scripts in the QMLA codebase consist of many configuration settings for running QMLA; only the lines in snippet in Listing A.1 need to be set according to altered to retrieve the corresponding figures. Note that the runtime of QMLA grows quite quickly with N_e, N_p (except for the AnalyticalLikelihood ES), especially for the entire QMLA algorithm; running QHL is feasible on a personal computer in < 30 minutes for $N_e = 1000; N_p = 3000$.

```
#!/bin/bash

###############
# QMLA run configuration
###############
num_instances=1
run_qhl=1 # perform QHL on known (true) model
run_qhl_mulit_model=0 # perform QHL for defined list of models.
exp=200 # number of experiments
prt=1000 # number of particles

###############
# QMLA settings
###############
plot_level=6
debug_mode=0

###############
# Choose an exploration strategy
###############

exploration_strategy='AnalyticalLikelihood'
```

Listing A.1: QMLA Launch scipt

Figure	Exploration Strategy	N_E	N_P	Data
Fig. 1.3	DemoHeuristicPGH	1000	3000	Nov_27/19_39
	DemoHeuristicNineEighths	1000	3000	Nov_27/19_40
	DemoHeuristicTimeList	1000	3000	Nov_27/19_42
	DemoHeuristicRandom	1000	3000	Nov_27/19_47
Fig. 1.4	DemoProbesPlus	1000	3000	Nov_27/14_43
	DemoProbesZero	1000	3000	Nov_27/14_45
	DemoProbesTomographic	1000	3000	Nov_27/14_46
	DemoProbes	1000	3000	Nov_27/14_47
Fig. 1.5	DemoProbesPlus	1000	3000	Nov_27/14_43
	DemoProbesZero	1000	3000	Nov_27/14_45
	DemoProbesTomographic	1000	3000	Nov_27/14_46
	DemoProbes	1000	3000	Nov_27/14_47
Fig. 1.2	AnalyticalLikelihood	500	2000	Nov_16/14_28
??	DemoIsing	500	5000	Nov_18/13_56
??	DemoIsing	1000	5000	Nov_18/13_56
??	DemoIsing	1000	5000	Nov_18/13_56
??	IsingLatticeSet	1000	4000	Nov_19/12_04
	IsingLatticeSet	1000	4000	Nov_19/12_04
	IsingLatticeSet	1000	4000	Nov_19/12_04
??	IsingLatticeSet	1000	4000	Sep_30/22_40
	HeisenbergLatticeSet	1000	4000	Oct_22/20_45
	FermiHubbardLatticeSet	1000	4000	Oct_02/00_09

Table A.1: Implementation details for figures used in the main text. Continued in Table A.2.

		N_E	N_P	Data
Figure	Exploration Strategy			
??	DemoBayesFactorsByFscore	500	2500	Dec_09/12_29
	DemoFractionalResourcesBayesFactorsByFscore	500	2500	Dec_09/12_31
	DemoBayesFactorsByFscore	1000	5000	Dec_09/12_33
	DemoBayesFactorsByFscoreEloGraphs	500	2500	Dec_09/12_32
??	HeisenbergGeneticXYZ	500	2500	Dec_10/14_40
??	HeisenbergGeneticXYZ	500	2500	Dec_10/14_40
	HeisenbergGeneticXYZ	500	2500	Dec_10/14_40
??	HeisenbergGeneticXYZ	500	2500	Dec_10/16_12
	HeisenbergGeneticXYZ	500	2500	Dec_10/16_12
??	NVCentreExperimentalData	1000	3000	2019/Oct_02/18_01
	SimulatedExperimentNVCentre	1000	3000	2019/Oct_02/18_16
??	NVCentreExperimentalData	1000	3000	2019/Oct_02/18_01
??	SimulatedExperimentNVCentre	1000	3000	2019/Oct_02/18_16
??	SimulatedExperimentNVCentre	1000	3000	2019/Oct_02/18_16
	NVCentreExperimentalData	1000	3000	2019/Oct_02/18_01
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_09/12_00
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_08/23_58
??	NVCentreGenticAlgorithmPrelearnedParameters	2	5	Sep_08/23_58

Table A.2: [Continued from Table A.1] Implementation details for figures used in the main text.

B FUNDAMENTALS

There are a number of concepts which are fundamental to any discussion of QM, but are likely to be known to most readers, and are therefore cumbersome to include in the main body of the thesis. We include them here for completeness[1].

B.1 LINEAR ALGEBRA

Here we review the language of linear algebra and summarise the basic mathematical techniques used throughout this thesis. We will briefly recall some definitions for reference.

- Notation

Definition of	Representation
Vector (or *ket*)	$\lvert\psi\rangle$
Dual Vector (or *bra*)	$\langle\psi\rvert$
Tensor Product	$\lvert\psi\rangle \otimes \lvert\phi\rangle$
Complex conjugate	$\lvert\psi^*\rangle$
Transpose	$\lvert\psi\rangle^T$
Adjoint	$\lvert\psi\rangle^\dagger = (\lvert\psi\rangle^*)^T$

Table B.1: Linear algebra definitions.

The dual vector of a vector (ket) $|\psi\rangle$ is given by $\langle\psi| = |\psi\rangle^\dagger$.

The *adjoint* of a matrix replaces each matrix element with its own complex conjugate, and then switches its columns with rows.

$$M^\dagger = \begin{pmatrix} M_{0,0} & M_{0,1} \\ M_{1,0} & M_{1,1} \end{pmatrix}^\dagger = \begin{pmatrix} M^*_{0,0} & M^*_{0,1} \\ M^*_{1,0} & M^*_{1,1} \end{pmatrix}^T = \begin{pmatrix} M^*_{0,0} & M^*_{1,0} \\ M^*_{0,1} & M^*_{1,1} \end{pmatrix} \tag{B.1}$$

1 Much of this description is reproduced from my undergraduate thesis [21].

The *inner product* of two vectors, $|\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{pmatrix}$ and $|\phi\rangle = \begin{pmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_n \end{pmatrix}$ is given by

$$\langle\phi|\psi\rangle = (|\phi\rangle^\dagger)\,|\psi\rangle = (\phi_1^* \; \phi_2^* \; \dots \; \phi_n^*) \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{pmatrix} = \phi_1^*\psi_1 + \phi_2^*\psi_2 + \cdots + \phi_n^*\psi_n \tag{B.2}$$

$|\psi\rangle_i, |\phi\rangle_i$ are complex numbers, and therefore the above is simply a sum of products of complex numbers. The inner product is often called the *scalar product*, which is in general complex.

B.2 POSTULATES OF QUANTUM MECHANICS

There are numerous statements of the postulates of quantum mechanics. Each version of the statements aims to achieve the same foundation, so we endeavour to explain them in the simplest terms.

1 Every moving particle in a conservative force field has an associated wave-function, $|\psi\rangle$. From this wave-function, it is possible to determine all physical information about the system.

2 All particles have physical properties called observables (denoted Q). In order to determine a value, Q, for a particular observable, there is an associated *operator* $\hat{Q}$, which, when acting on the particles wavefunction, yields the value times the wavefunction. The observable Q is then the eigenvalue of the operator $\hat{Q}$.

$$\hat{Q}\,|\psi\rangle = q|\psi\rangle \tag{B.3}$$

3 Any such operator $\hat{Q}$ is Hermitian

$$\hat{Q}^\dagger = \hat{Q} \tag{B.4}$$

4 The set of eigenfunctions for any operator $\hat{Q}$ forms a complete set of linearly independent functions.

5 For a system with wavefunction $|\psi\rangle$, the expectation value of an observable Q with respect to an operator $\hat{Q}$ is denoted by $\langle q\rangle$ and is given by

$$\langle q\rangle = \langle\psi|\hat{Q}|\psi\rangle \tag{B.5}$$

6 The time evolution of $|\psi\rangle$ is given by the time dependent *Schrodinger Equation*

$$i\hbar\frac{\partial\psi}{\partial t} = \hat{H}\psi, \tag{B.6}$$

where $\hat{H}$ is the system's Hamiltonian.

Using these building blocks, we can begin to construct a language to describe quantum systems.

B.3 STATES

An orhthonormal basis consists of vectors of unit length which do not overlap, e.g. $|x_1\rangle = \begin{pmatrix}1\\0\end{pmatrix}, |x_2\rangle = \begin{pmatrix}0\\1\end{pmatrix} \Rightarrow \langle x_1|x_2\rangle = 0$. In general, if $\{|x\rangle\}$ are the eigenstates of a system, then the system can be written as some state vector, $|\psi\rangle$, in general a superposition over the basis-vectors:

$$|\psi\rangle = \sum_x a_x|x\rangle \tag{B.7a}$$

$$\text{subject to} \quad \sum_x |a_x|^2 = 1, \quad a_x \in \mathbb{C} \tag{B.7b}$$

The *state space* of a physical system (classical or quantum) is then the set of all possible states the system can exist in, i.e the set of all possible values for $|\psi\rangle$ such that Eq. (B.7b) are satisfied.

For example, photons can be polarised horizontally ($\leftrightarrow$) or vertically ($\updownarrow$); take those two conditions as observable states to define the eigenstates of a two-level system, so we can designate the photon as a qubit. Then we can map the two states to a 2-dimensional, x-y plane: a general vector on such a plane can be represented by a vector with coordinates $\begin{pmatrix}x\\y\end{pmatrix}$. These polarisations can then be thought of as standard basis vectors in linear algebra. Denote $\leftrightarrow$ as the eigenstate $|0\rangle$ and $\updownarrow$ as $|1\rangle$

$$|\leftrightarrow\rangle = |0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \qquad \text{A unit vector along x-axis} \tag{B.8a}$$

$$|\updownarrow\rangle = |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \qquad \text{A unit vector along y-axis} \tag{B.8b}$$

Now, in relation to the concept of superposition, we can consider, for example, a photon in an even superposition of the vertical and horizontal polarisations, evenly splitting the two basis vectors. As such, we would require that, upon measurement, it is equally likely that the photon will *collapse* into the polarised state along x as it is to collapse along y. That is, we want $\Pr(\updownarrow) = \Pr(\leftrightarrow)$ so assign equal modulus amplitudes to the two possibilities:

$$|\psi\rangle = a\,|\leftrightarrow\rangle + b\,|\updownarrow\rangle\,, \quad \text{with} \quad \Pr(\updownarrow) = \Pr(\leftrightarrow) \Rightarrow |a|^2 = |b|^2 \tag{B.9}$$

We consider here a particular case, due to the significance of the resultant basis, where $\leftrightarrow$-polarisation and $\updownarrow$-polarisation have real amplitudes $a, b \in \mathbb{R}$.

$$\begin{aligned} &\Rightarrow a = \pm b \quad \text{but also} \quad |a|^2 + |b|^2 = 1 \\ \Rightarrow \quad & a = \frac{1}{\sqrt{2}} \quad ; \quad b = \pm\frac{1}{\sqrt{2}} \\ &\Rightarrow |\psi\rangle = \frac{1}{\sqrt{2}}\,|\leftrightarrow\rangle \pm \frac{1}{\sqrt{2}}\,|\updownarrow\rangle \\ &\Rightarrow |\psi\rangle = \frac{1}{\sqrt{2}}\,|0\rangle \pm \frac{1}{\sqrt{2}}\,|1\rangle \end{aligned} \tag{B.10}$$

These particular superpositions are of significance:

$$|+\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right) \tag{B.11a}$$

$$|-\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right) \tag{B.11b}$$

This is called the Hadamard basis: it is an equally valid vector space as the standard basis which is spanned by $\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}$, as it is simply a rotation of the standard basis.

B.3.1 *Mulitpartite systems*

In reality, we often deal with systems of multiple particles, represented by multiple qubits. Mathematically, we consider the state vector of a system containing n qubits as being the tensor product of the n qubits' individual state vectors[2]. For instance, suppose a 2-qubit system, $|\psi\rangle$ consisting of two independent qubits $|\psi_A\rangle$ and $|\psi_B\rangle$:

$$|\psi\rangle = |\psi_A\rangle\,|\psi_B\rangle = |\psi_A\psi_B\rangle = |\psi_A\rangle \otimes |\psi_B\rangle \tag{B.12}$$

Consider first a simple system of 2 qubits. Measuring in the standard basis, these qubits will have to collapse in to one of the basis states $|0,0\rangle, |0,1\rangle, |1,0\rangle, |1,1\rangle$. Thus, for such a 2-qubit system, we have the general superposition

$$|\psi\rangle = \alpha_{0,0}|0,0\rangle + \alpha_{0,1}|0,1\rangle + \alpha_{1,0}|1,0\rangle + \alpha_{1,1}|1,1\rangle$$

2 We will later discuss entangled states, which can not be described thus.

where $\alpha_{i,j}$ is the amplitude for measuring the system as the state $|i,j\rangle$. This is perfectly analogous to a classical 2-bit system necessarily occupying one of the four possibilities $\{(0,0),(0,1),(1,0),(1,1)\}$.

Hence, for example, if we wanted to concoct a two-qubit system composed of one qubit in the state $|+\rangle$ and one in $|-\rangle$

$$\begin{aligned}
|\psi\rangle &= |+\rangle \otimes |-\rangle \\
|\psi\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\
&= \frac{1}{2}\left[|00\rangle - |01\rangle + |10\rangle - |11\rangle\right] \\
&= \frac{1}{2}\left[\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} - \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} - \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix}\right] \\
&= \frac{1}{2}\left[\begin{pmatrix}1\\0\\0\\0\end{pmatrix} - \begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \begin{pmatrix}0\\0\\1\\0\end{pmatrix} - \begin{pmatrix}0\\0\\0\\1\end{pmatrix}\right]. \\
\Rightarrow |\psi\rangle &= \frac{1}{2}\begin{pmatrix}1\\-1\\-1\\1\end{pmatrix}
\end{aligned} \tag{B.13}$$

That is, the two qubit system – and indeed any two qubit system – is given by a linear combination of the four basis vectors

$$\{|00\rangle, |0,1\rangle, |10\rangle, |11\rangle\} = \left\{\begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}\right\}. \tag{B.14}$$

We can notice that a single qubit system can be described by a linear combination of two basis vectors, and that a two qubit system requires four basis vectors to describe it. In general we can say that an n-qubit system is represented by a linear combination of 2^n basis vectors.

B.3.2 *Registers*

A *register* is generally the name given to an array of controllable quantum systems; here we invoke it to mean a system of multiple qubits, specifically a subset of the total number of available qubits. For example, a register of ten qubits can be denoted $|x[10]\rangle$, and we can think of the system as a register of six qubits together with a register of three and another register of one qubit.

$$|x[10]\rangle = |x_1[6]\rangle \otimes |x_2[3]\rangle \otimes |x_3[1]\rangle$$

B.4 ENTANGLEMENT

Another unique property of quantum systems is that of *entanglement*: when two or more particles interact in such a way that their individual quantum states can not be described independent of the other particles. A quantum state then exists for the system as a whole instead. Mathematically, we consider such entangled states as those whose state can not be expressed as a tensor product of the states of the individual qubits it's composed of: they are dependent upon the other.

To understand what we mean by this dependence, consider a counter-example. Consider the Bell state,

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right), \tag{B.15}$$

if we measure this state, we expect that it will be observed in either eigenstate $|00\rangle$ or $|11\rangle$, with equal probability due to their amplitudes' equal magnitudes. The bases for this state are simply the standard bases, $|0\rangle$ and $|1\rangle$. Thus, according to our previous definition of systems of multiple qubits, we would say this state can be given as a combination of two states, like Eq. (B.12),

$$\begin{aligned} |\Phi^+\rangle &= |\psi_1\rangle \otimes |\psi_2\rangle \\ &= (a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle) \\ &= a_1a_2|00\rangle + a_1b_2|01\rangle + b_1a_2|10\rangle + b_1b_2|11\rangle \end{aligned} \tag{B.16}$$

However we require $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, which would imply $a_1b_2 = 0$ and $b_1a_2 = 0$. These imply that either $a_1 = 0$ or $b_2 = 0$, and also that $b_1 = 0$ or a_2=0, which are obviously invalid since we require that $a_1a_2 = b_1b_2 = \frac{1}{\sqrt{2}}$. Thus, we cannot express $|\Phi^+\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$; this inability to separate the first and second qubits is what we term *entanglement*.

B.5 UNITARY TRANSFORMATIONS

A fundamental concept in quantum mechanics is that of performing transformations on states. *Quantum transformations*, or *quantum operators*, map a quantum state into a new state within the same Hilbert space. There are certain restrictions on a physically possible quantum transformation: in order that U is a valid transformation acting on some superposition $|\psi\rangle = a_1|\psi_1\rangle + a_2|\psi_2\rangle + \ldots a_k|\psi_k\rangle$, U must be linear

$$U(a_1|\psi_1\rangle + a_2|\psi_2\rangle + \ldots a_k|\psi_k\rangle) = a_1(U|\psi_1\rangle) + a_2(U|\psi_2\rangle) + \cdots + a_k(U|\psi_k\rangle). \tag{B.17}$$

To fulfil these properties, we require that U *preserve the inner product*:

$$\langle \psi_0 | U^\dagger U | \psi \rangle = \langle \psi_0 | \psi \rangle$$

That is, we require that any such transformation be *unitary*:

$$UU^\dagger = I \Rightarrow U^\dagger = U^{-1} \tag{B.18}$$

Unitarity is a sufficient condition to describe any valid quantum operation: any quantum transformation can be described by a unitary transformation, and any unitary transformation corresponds to a physically implementable quantum transformation.

Then, if U_1 is a unitary transformation that acts on the space $\mathcal{H}_1$ and U_2 acts on $\mathcal{H}_2$, the product of the two unitary transformations is also unitary. The tensor product $U_1 \otimes U_2$ acts on the space $\mathcal{H}_1 \otimes \mathcal{H}_2$. So, then, supposing a system of two separable qubits, $|\psi_1\rangle$ and $|\psi_2\rangle$ where we wish to act on $|\psi_1\rangle$ with operator U_1 and on $|\psi_2\rangle$ with U_2, we perform it as

$$(U_1 \otimes U_2)\left(|\psi_1\rangle \otimes |\psi_2\rangle\right) = \left(U_1 |\psi_1\rangle\right) \otimes \left(U_2 |\psi_2\rangle\right) \tag{B.19}$$

B.6 DIRAC NOTATION

In keeping with standard practice, we employ *Dirac notation* throughout this thesis. Vectors are denoted by *kets* of the form $|a\rangle$. For example, the standard basis is represented by,

$$\begin{aligned} |x\rangle = |0\rangle &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ |y\rangle = |1\rangle &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{aligned} \tag{B.20}$$

We saw in Table B.1 that for every such ket, $|\psi\rangle$, there exists a *dual vector*: its complex conjugate transpose, called the *bra* of such a vector, denoted $\langle\psi|$. That is,

$$\begin{aligned} \langle\psi|^\dagger &= |\psi\rangle \\ |\psi\rangle^\dagger &= \langle\psi| \end{aligned} \tag{B.21}$$

$$|\psi\rangle = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_n \end{pmatrix} \Rightarrow \langle\psi| = \begin{pmatrix} \psi_1^* & \psi_2^* & \dots & \psi_n^* \end{pmatrix} \tag{B.22}$$

Then if we have two vectors $|\psi\rangle$ and $|\phi\rangle$, their *inner product* is given as $\langle\psi|\phi\rangle = \langle\phi|\psi\rangle$.

$$
\begin{aligned}
|\psi\rangle = \begin{pmatrix}\psi_1\\ \psi_2\\ \psi_3\\ \vdots\\ \psi_n\end{pmatrix} \quad ; \quad |\phi\rangle = \begin{pmatrix}\phi_1\\ \phi_2\\ \phi_3\\ \vdots\\ \phi_n\end{pmatrix}& \\
\Rightarrow \langle\phi| = \begin{pmatrix}\phi_1^* & \phi_2^* & \phi_3^* & \dots & \phi_n^*\end{pmatrix}& \\
\Rightarrow \langle\phi|\,|\psi\rangle = \begin{pmatrix}\phi_1^* & \phi_2^* & \phi_3^* & \dots & \phi_n^*\end{pmatrix}\begin{pmatrix}\psi_1\\ \psi_2\\ \psi_3\\ \vdots\\ \psi_n\end{pmatrix}& \\
\Rightarrow \langle\phi|\,|\psi\rangle = \phi_1^*\psi_1 + \phi_2^*\psi_2 + \phi_3^*\psi_3 + \cdots + \phi_n^*\psi_n&
\end{aligned}
\tag{B.23}
$$

Example B.6.1.

$$
\begin{aligned}
|\psi\rangle = \begin{pmatrix}1\\2\\3\end{pmatrix} \quad ; \quad |\phi\rangle = \begin{pmatrix}4\\5\\6\end{pmatrix}& \\
\Rightarrow \langle\phi|\,|\psi\rangle = \begin{pmatrix}4 & 5 & 6\end{pmatrix}\begin{pmatrix}1\\2\\3\end{pmatrix}& \\
= (4)(1) + (5)(2) + (6)(3) = 32&
\end{aligned}
\tag{B.24}
$$

Similarly, their *outer product* is given as $|\phi\rangle\,\langle\psi|$. Multiplying a column vector by a row vector thus gives a matrix. Matrices generated by a outer products then define operators:

Example B.6.2.

$$
\begin{pmatrix}1\\2\end{pmatrix}\begin{pmatrix}3 & 4\end{pmatrix} = \begin{pmatrix}3 & 4\\ 6 & 8\end{pmatrix}
\tag{B.25}
$$

Then we can say, for $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$

$$
|0\rangle\,\langle 0| = \begin{pmatrix}1 & 0\\ 0 & 0\end{pmatrix}
\tag{B.26a}
$$

$$|0\rangle\langle 1| = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \tag{B.26b}$$

$$|1\rangle\langle 0| = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \tag{B.26c}$$

$$|1\rangle\langle 1| = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{B.26d}$$

And so any 2-dimensional linear transformation in the standard basis $|0\rangle, |1\rangle$ can be given as a sum

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = a\,|0\rangle\langle 0| + b\,|0\rangle\langle 1| + c\,|1\rangle\langle 0| + d\,|1\rangle\langle 1| \tag{B.27}$$

This is a common method of representing operators as outer products of vectors. A transformation that *exchanges* a particle between two states, say $|0\rangle \leftrightarrow |1\rangle$ is given by the operation

$$\hat{Q} : \begin{cases} |0\rangle \to |1\rangle \\ |1\rangle \to |0\rangle \end{cases}$$

Which is equivalent to the outer product representation

$$\hat{Q} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

For clarity, here we will prove this operation

Example B.6.3.

$$\begin{aligned} \hat{Q} &= |0\rangle\langle 1| + |1\rangle\langle 0| \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{aligned}$$

So then, acting on $|0\rangle$ and $|1\rangle$ gives

$$\hat{Q}\,|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

$$\hat{Q}\left|1\right\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \left|0\right\rangle$$

To demonstrate how Dirac notation simplifies this:

$$\hat{Q}\left|0\right\rangle = (\left|0\right\rangle\left\langle 1\right| + \left|1\right\rangle\left\langle 0\right|)\left|0\right\rangle$$

$$= \left|0\right\rangle\left\langle 1\right|\left|0\right\rangle + \left|1\right\rangle\left\langle 0|0\right\rangle$$
$$= \left|0\right\rangle\left\langle 1|0\right\rangle + \left|1\right\rangle\left\langle 0|0\right\rangle$$

Then, since $\left|0\right\rangle$ and $\left|1\right\rangle$ are orthogonal basis, their inner product is 0 and the inner product of a vector with itself is 1, $(\langle 1|1\rangle = \langle 0|0\rangle = 1, \langle 0|1\rangle = \langle 1|0\rangle = 0)$. So,

$$\begin{aligned} \hat{Q}\left|0\right\rangle &= \left|0\right\rangle(0) + \left|1\right\rangle(1) \\ \Rightarrow \hat{Q}\left|0\right\rangle &= \left|1\right\rangle \end{aligned} \tag{B.28}$$

And similarly for $\hat{Q}\left|1\right\rangle$. This simple example then shows why Dirac notation can significantly simplify calculations across quantum mechanics, compared to standard matrix and vector notation. To see this more clearly, we will examine a simple 2-qubit state under such operations. The method generalises to operating on two or more qubits generically: we can define any operator which acts on two qubits as a sum of outer products of the basis vectors $\left|00\right\rangle, \left|01\right\rangle, \left|10\right\rangle$ and $\left|11\right\rangle$. We can similarly define any operator which acts on an n qubit state as a linear combination of the 2^n basis states generated by the n qubits.

Example B.6.4. *To define a transformation that will exchange basis vectors $\left|00\right\rangle$ and $\left|11\right\rangle$, while leaving $\left|01\right\rangle$ and $\left|10\right\rangle$ unchanged (ie exchanging $\left|01\right\rangle \leftrightarrow \left|01\right\rangle, \left|10\right\rangle \leftrightarrow \left|10\right\rangle$) we define an operator*

$$\hat{Q} = \left|00\right\rangle\left\langle 11\right| + \left|11\right\rangle\left\langle 00\right| + \left|10\right\rangle\left\langle 10\right| + \left|01\right\rangle\left\langle 01\right| \tag{B.29}$$

Then, using matrix calculations this would require separately calculating the four outer products in the above sum and adding them to find a 4×4 matrix to represent $\hat{Q}$, which then acts on a state $\left|\psi\right\rangle$. Instead, consider first that $\left|\psi\right\rangle = \left|00\right\rangle$, ie one of the basis vectors our transformation is to change:

$$\hat{Q}\left|00\right\rangle = (\left|00\right\rangle\left\langle 11\right| + \left|11\right\rangle\left\langle 00\right| + \left|10\right\rangle\left\langle 10\right| + \left|01\right\rangle\left\langle 01\right|)\ \left|00\right\rangle \tag{B.30}$$

And as before, only the inner products of a vector with itself remains:

$$\begin{aligned} &= \left|00\right\rangle\left\langle 11\right|\left.00\right\rangle + \left|11\right\rangle\left\langle 00\right|\left.00\right\rangle + \left|10\right\rangle\left\langle 10\right|\left.00\right\rangle + \left|01\right\rangle\left\langle 01\right|\left.00\right\rangle \\ &= \left|00\right\rangle(0) + \left|11\right\rangle(1) + \left|10\right\rangle(0) + \left|01\right\rangle(0) \\ \Rightarrow &\hat{Q}\left|00\right\rangle = \left|11\right\rangle \end{aligned} \tag{B.31}$$

i.e the transformation has performed $\hat{Q} : |00\rangle \rightarrow |11\rangle$ as expected. Then, if we apply the same transformation to a state which does not depend on one of the target states, eg,

$$\begin{aligned}
|\psi\rangle =& a\,|10\rangle + b\,|01\rangle \\
\hat{Q}\,|\psi\rangle =& \Big(\, |00\rangle\langle 11| + |11\rangle\langle 00| + |10\rangle\langle 10| + |01\rangle\langle 01| \Big)\Big(a\,|10\rangle + b\,|01\rangle\,\Big) \\
=& a\Big(\, |00\rangle\langle 11|\,|10\rangle + |11\rangle\langle 00|\,|10\rangle + |10\rangle\langle 10|\,|10\rangle + |01\rangle\langle 01|\,|10\rangle\,\Big) \\
&+ b\Big(\, |00\rangle\langle 11|\,|01\rangle + |11\rangle\langle 00|\,|01\rangle + |10\rangle\langle 10|\,|01\rangle + |01\rangle\langle 01|\,|01\rangle\,\Big)
\end{aligned} \tag{B.32}$$

And since the inner product is a scalar, we can factor terms such as $\langle 11|10\rangle$ to the beginning of expressions, eg $|00\rangle\langle 11|\,|10\rangle = \langle 11\,|10\rangle\,|00\rangle$, and we also know

$$\begin{aligned}
\langle 11|10\rangle = \langle 00|10\rangle = \langle 01|10\rangle = \langle 11|01\rangle = \langle 00|01\rangle = \langle 10|01\rangle = 0 \\
\langle 10|10\rangle = \langle 01|01\rangle = 1
\end{aligned} \tag{B.33}$$

We can express the above as

$$\begin{aligned}
\hat{Q}\,|\psi\rangle =& a\Big((0)\,|00\rangle + (0)\,|11\rangle + (1)\,|10\rangle + (0)\,|01\rangle\,\Big) \\
&+b\Big((0)\,|00\rangle + (0)\,|11\rangle + (0)\,|10\rangle + (1)|01\rangle\Big) \\
=& a\,|10\rangle\,| + b|01\rangle \\
=& \,|\psi\rangle
\end{aligned} \tag{B.34}$$

Then it is clear that, when $|\psi\rangle$ is a superposition of states unaffected by transformation $\hat{Q}$, then $\hat{Q}\,|\psi\rangle = |\psi\rangle$.

This method generalises to systems with greater numbers of particles (qubits). If we briefly consider a 3 qubit system - and initialise all qubits in the standard basis state $|0\rangle$ - then the system is represented by $|000\rangle = |0\rangle \otimes |0\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. This quantity is an 8-row vector. To calculate the outer product $\langle 000|000\rangle$, we would be multiplying an 8-column bra $\langle 000|$ by an 8-row ket $|000\rangle$. Clearly then we will be working with 8×8 matrices, which will become quite difficult to maintain effectively and efficiently quite fast. As we move to systems of larger size, standard matrix multiplication becomes impractical for hand-written analysis, although of course remains tractable computationally up to $n \sim 10$ qubits. It is obvious that Dirac's bra/ket notation is a helpful, pathematically precise tool for QM.

C EXAMPLE EXPLORATION STRATEGY RUN

Here we provide a complete example of how to run the Quantum Model Learning Agent (QMLA) framework, including how to implement a custom exploration strategy (ES), and generate/interpret analysis. Note: these examples are included in the QMLA documentation in a format that may be easier to follow – where possible, we recommend readers follow the Tutorial section of [22].

First, *fork* the QMLA codebase from [23] to a Github user account (referred to as username in Listing C.1). Now, we must download the code base and ensure it runs properly; these instructions are implemented via the command line[1].

The steps of preparing the codebase are

1. install redis;
2. create a virtual Python environment for installing QMLA dependencies without damaging other parts of the user's environment;
3. download the QMLA codebase from the forked Github repository;
4. install packages upon which QMLA depends.

```
# Install redis (database broker)
sudo apt update
sudo apt install redis-server

# make directory for QMLA
cd
mkdir qmla_test
cd qmla_test

# make Python virtual environment for QMLA
# note: change Python3.6 to desired version
sudo apt-get install python3.6-venv
python3.6 -m venv qmla-env
source qmla-env/bin/activate
```

1 Note: these instructions are tested for Linux and presumed to work on Mac, but untested on Windows. It is likely some of the underlying software (redis servers) can not be installed on Windows, so running on *Windows Subsystem for Linux* is advised.

```
# Download QMLA
git clone --depth 1 https://github.com/username/QMLA.git #
    REPLACE username

# Install dependencies
cd QMLA
pip install -r requirements.txt
```

Listing C.1: QMLA codebase setup language

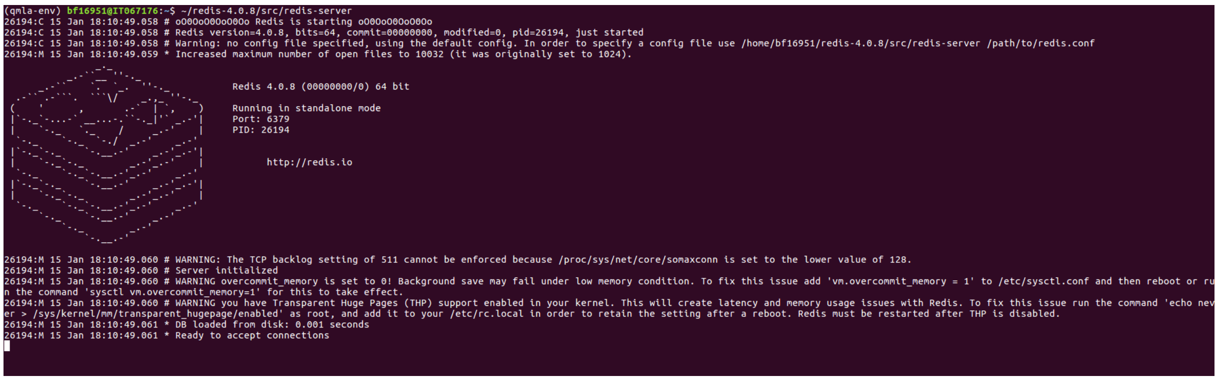
Note there may be a problem with some packages in the requirements.txt arising from the attempt to install them all through a single call to pip install. Ensure these are all installed before proceeding.

When all of the requirements are installed, test that the framework runs. QMLA uses redis databases to store intermittent data: we must manually initialise the database. Run the following (note: here we list redis-4.0.8, but this must be corrected to reflect the version installed on the user's machine in the above setup section):

```
~/redis-4.0.8/src/redis-server
```

Listing C.2: Launch redis database

which should give something like Fig. C.1.



```
(qmla-env) bf16951@IT067176:~$ ~/redis-4.0.8/src/redis-server
26194:C 15 Jan 18:10:49.058 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
26194:C 15 Jan 18:10:49.058 # Redis version=4.0.8, bits=64, commit=00000000, modified=0, pid=26194, just started
26194:C 15 Jan 18:10:49.058 # Warning: no config file specified, using the default config. In order to specify a config file use /home/bf16951/redis-4.0.8/src/redis-server /path/to/redis.conf
26194:M 15 Jan 18:10:49.059 * Increased maximum number of open files to 10032 (it was originally set to 1024).

                                Redis 4.0.8 (00000000/0) 64 bit

                                Running in standalone mode
                                Port: 6379
                                PID: 26194

                                      http://redis.io

26194:M 15 Jan 18:10:49.060 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
26194:M 15 Jan 18:10:49.060 # Server initialized
26194:M 15 Jan 18:10:49.060 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or ru
n the command 'sysctl vm.overcommit_memory=1' for this to take effect.
26194:M 15 Jan 18:10:49.060 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo nev
er > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
26194:M 15 Jan 18:10:49.061 * DB loaded from disk: 0.001 seconds
26194:M 15 Jan 18:10:49.061 * Ready to accept connections
```

Figure C.1: Terminal running redis-server.

In a text editor, open qmla_test/QMLA/launch/local_launch.sh; here we will ensure that we are running the QHL algorithm, with 5 experiments and 20 particles, on the ES named TestInstall. Ensure the first few lines of local_launch.sh read:

```
#!/bin/bash

##### -------------------------------------------------- #####
# QMLA run configuration
##### -------------------------------------------------- #####
num_instances=2 # number of instances in run
run_qhl=0 # perform QHL on known (true) model
run_qhl_multi_model=0 # perform QHL for defined list of models
experiments=2 # number of experiments
particles=10 # number of particles
plot_level=5


##### -------------------------------------------------- #####
# Choose an exploration strategy
# This will determine how QMLA proceeds.
##### -------------------------------------------------- #####
exploration_strategy="TestInstall"
```

Listing C.3: local_launch script

Ensure the terminal running redis is kept active, and open a separate terminal window. We must activate the Python virtual environment configured for QMLA, which we set up in Listing C.1. Then, we navigate to the QMLA directory, and launch:

```

# activate the QMLA Python virtual environment
source qmla_test/qmla-env/bin/activate

# move to the QMLA directory
cd qmla_test/QMLA
# Run QMLA
cd launch
./local_launch.sh
```

Listing C.4: Launch QMLA

There may be numerous warnings, but they should not affect whether QMLA has succeeded; QMLA will raise any significant error. Assuming the run has completed successfully, QMLA stores the run's results in a subdirectory named by the date and time it was started. For example, if the run was initialised on January 1st at 01:23, navigate to the corresponding directory by

```
cd results/Jan_01/01_23
```

Listing C.5: QMLA results directory

For now it is sufficient to notice that the code has run successfully: it should have generated (in results/Jan_01/01_23) files like storage_001.p and results_001.p.

C.1 CUSTOM EXPLORATION STRATEGY

Next, we design a basic ES, for the purpose of demonstrating how to run the algorithm. ESs are placed in the directory qmla/exploration_strategies. To make a new one, navigate to the exploration strategies directory, make a new subdirectory, and copy the template file.

```
cd ~/qmla_test/QMLA/exploration_strategies/
mkdir custom_es

# Copy template file into example
cp template.py custom_es/example.py
cd custom_es
```

Listing C.6: QMLA codebase setup

Ensure QMLA will know where to find the ES by importing everything from the custom ES directory into to the main exploration_strategy module. Then, in the custom_es directory, make a file called __init__.py which imports the new ES from the example.py file. To add any further ESs inside the directory custom_es, include them in the custom __init__.py, and they will automatically be available to QMLA.

```
# inside qmla/exploration_strategies/custom_es
#   __init__.py
from qmla.exploration_strategies.custom_es.example import *

# inside qmla/exploration_strategies, add to the existing
# __init__.py
from qmla.exploration_strategies.custom_es import *
```

Listing C.7: Providing custom exploration strategy to QMLA

Now, change the structure (and name) of the ES inside custom_es/example.py. Say we wish to target the true model

$$
\begin{aligned}
\vec{\alpha} &= \begin{pmatrix} \alpha_{1,2} & \alpha_{2,3} & \alpha_{3,4} \end{pmatrix} \\
\vec{T} &= \begin{pmatrix} \hat{\sigma}_z^1 \otimes \hat{\sigma}_z^2 \\ \hat{\sigma}_z^2 \otimes \hat{\sigma}_z^3 \\ \hat{\sigma}_z^3 \otimes \hat{\sigma}_z^4 \end{pmatrix} \\
\implies \hat{H}_0 &= \hat{\sigma}_z^{(1,2)} \hat{\sigma}_z^{(2,3)} \hat{\sigma}_z^{(3,4)}
\end{aligned}
\tag{C.1}
$$

QMLA interprets models as strings, where terms are separated by $+$, and parameters are implicit. So the target model in Eq. (C.1) will be given by

pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4.

Adapting the template ES slightly, we can define a model generation strategy with a small number of hard coded candidate models introduced at the first branch of the exploration tree. We will also set the parameters of the terms which are present in $\hat{H}_0$, as well as the range in which to search parameters. Keeping the imports at the top of the example.py, rewrite the ES as:

```
class ExampleBasic(
    exploration_strategy.ExplorationStrategy
):

    def __init__(
        self,
        exploration_rules,
        true_model=None,
        **kwargs
    ):
        self.true_model = 'pauliSet_1J2_zJz_d4+
           pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4'
        super().__init__(
            exploration_rules=exploration_rules,
            true_model=self.true_model,
            **kwargs
        )

        self.initial_models = None
        self.true_model_terms_params = {
            'pauliSet_1J2_zJz_d4' : 2.5,
            'pauliSet_2J3_zJz_d4' : 7.5,
            'pauliSet_3J4_zJz_d4' : 3.5,
        }
        self.tree_completed_initially = True
        self.min_param = 0
        self.max_param = 10

    def generate_models(self, **kwargs):

        self.log_print(["Generating models; spawn step {}".format
           (self.spawn_step)])
        if self.spawn_step == 0:
            # chains up to 4 sites
            new_models = [
                'pauliSet_1J2_zJz_d4',
                'pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4',
                'pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+
                   pauliSet_3J4_zJz_d4',
            ]
            self.spawn_stage.append('Complete')

        return new_models
```

Listing C.8: ExampleBasic exploration strategy.

To run[2] the example ES for a meaningful test, return to the local_launch of Listing C.3, but change some of the settings:

```
particles=2000
experiments=500
run_qhl=1
exploration_strategy=ExampleBasic
```

Listing C.9: local_launch configuration for QHL.

Run locally again as in Listing C.4; then move to the results directory as in Listing C.5.

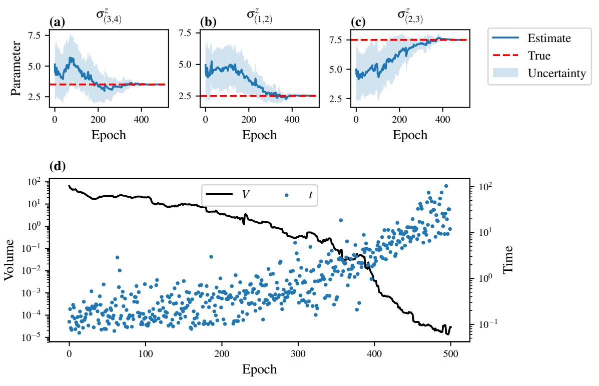
2 Note this will take up to 15 minutes to run. This can be reduced by lowering the values of particles, experiments, which is sufficient for testing but note that the outcomes will be less effective than those presented in the figures of this section.

C.2 ANALYSIS

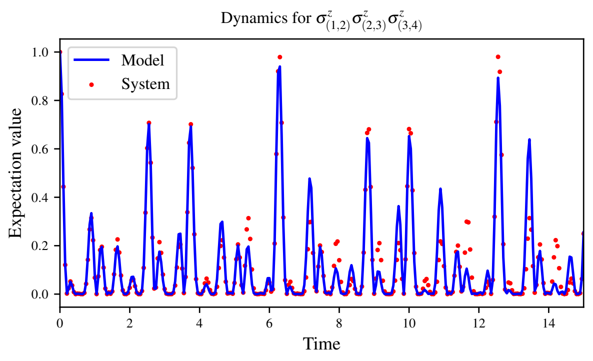
QMLA stores results and generates plots over the entire range of the algorithm[3], i.e. the run, instance and models. The depth of analysis performed automatically is set by the user control plot_level in local_launch.sh; for plot_level=1, only the most crucial figures are generated, while plot_level=6 generates plots for every individual model considered. For model searches across large model spaces and/or considering many candidates, excessive plotting can cause considerable slow-down, so users should be careful to generate plots only to the degree they will be useful. Next we show some examples of the available plots.

C.2.1 *Model analysis*

We have just run quantum Hamiltonian learning (QHL) for the model in Eq. (C.1) for a single instance, using a reasonable number of particles and experiments, so we expect to have trained the model well. Instance-level results are stored (e.g. for the instance with qmla_id=1) in Jan_01/01_23/instances/qmla_1. Individual models' insights can be found in model_training, e.g. the model's learning_summary Fig. C.2a, and dynamics in Fig. C.2b.



(a)



(b)

Figure C.2: Model analysis plots, stored in (for example) Jan_01/01_23/instances/qmla_1/model_training. **a** learning_summary_1. Displays the outcome of QHL for the given model: Subfigures (a)-(c) show the estimates of the parameters; (d) shows the total parameterisation volume against experiments trained upon, along with the evolution times used for those experiments. **(b)** dynamics_1 The model's attempt at reproducing dynamics from $\hat{H}_0$.

3 Recall that a single implementation of QMLA is called an instance, while a series of instances – which share the same target model – is called the run.

C.2.2 *Instance analysis*

Now we can run the full QMLA algorithm, i.e. train several models and determine the most suitable. QMLA will call the generate_models method of the ExampleBasic ES, set in Listing C.8, which tells QMLA to construct three models on the first branch, then terminate the search. Here we need to train and compare all models so it takes considerably longer to run: for the purpose of testing, we reduce the resources so the entire algorithm runs in about 15 minutes. Some applications will require significantly more resources to learn effectively. In realistic cases, these processes are run in parallel, as we will cover in Appendix C.3.

Reconfigure a subset of the settings in the local_launch.sh script (Listing C.3) and run it again:

```
experiments=250
particles=1000
run_qhl=0
exploration_strategy=ExampleBasic
```

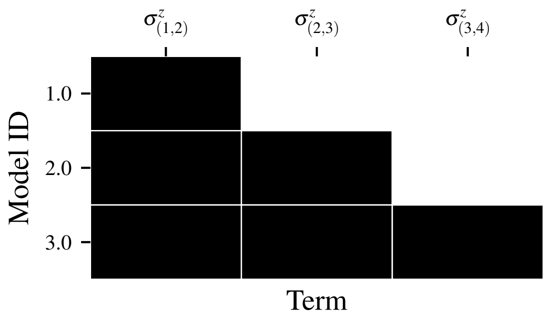
Listing C.10: local_launch configuration for QMLA.

In the corresponding results directory, navigate to instances/qmla_1, where instance level analysis are available.

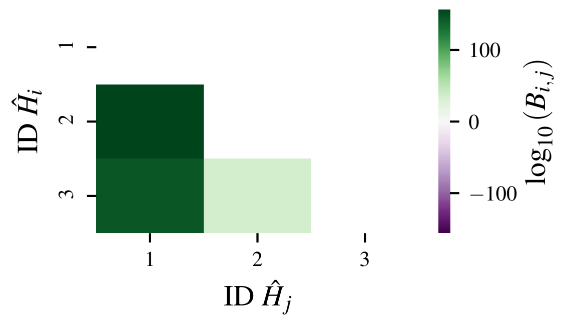
```
cd results/Jan_01/01_23/instances/qmla_1
```

Listing C.11: Navigating to instance results.

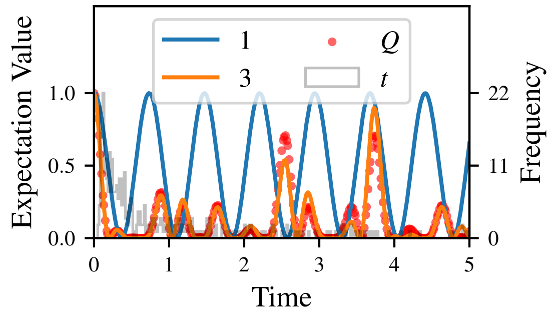
Figures of interest here show the composition of the models (Fig. C.3a), as well as the Bayes factors between candidates (Fig. C.3b). Individual model comparisons – i.e. Bayes factor (BF) – are shown in Fig. C.3c, with the dynamics of all candidates shown in Fig. C.4c. The probes used during the training of all candidates are also plotted (Fig. C.3e).



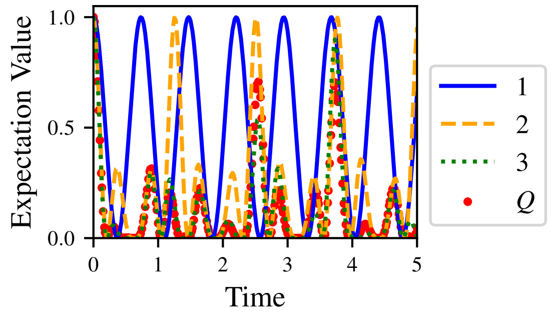
(a)



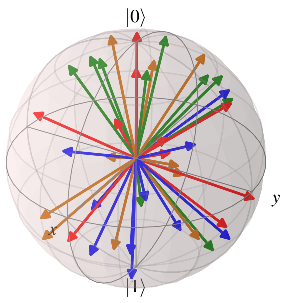
(b)



(c)



(d)



(e)

Figure C.3: QMLA plots; found within instance directory e.g. Jan_01/01_23/instances/qmla_1, and its subdirectories. **(a)** composition_of_models: constituent terms of all considered models, indexed by their model IDs. Here model 3 is $\hat{H}_0$. **(b)** bayes_factors: Bayes factor (BF) comparisons between all models. Bayes factors (BFs) are read as $B_{i,j}$ where i is the model with lower ID, e.g. $B_{1,2}$ rather than $B_{2,1}$. Thus $B_{ij} > 0$ (< 0) indicates $\hat{H}_i$ $(\hat{H}_j)$, i.e. the model on the y-axis (x-axis) is the stronger model. **(c)** comparisons/BF_1_3: direct comparison between models with IDs 1 and 3, showing their reproduction of the system dynamics (red dots, Q), as well as the times (experiments) against which the BF was calculated. **(d)** branches/dynamics_branch_1: dynamics of all models considered on the branch compared with system dynamics (red dots, Q). **(e)** probes_bloch_sphere: probes used for training models in this instance (only showing 1-qubit versions).

C.2.3 *Run analysis*

Considering a number of instances together is a *run*. In general, this is the level of analysis of most interest: an individual instance is liable to errors due to the probabilistic nature of the model training and generation subroutines. On average, however, we expect those elements to perform well, so across a significant number of instances, we expect the average outcomes to be meaningful.

Each results directory has an analyse.sh script to generate plots at the run level.

```
cd results/Jan_01/01_23
./analyse.sh
```

Listing C.12: Analysing QMLA run.

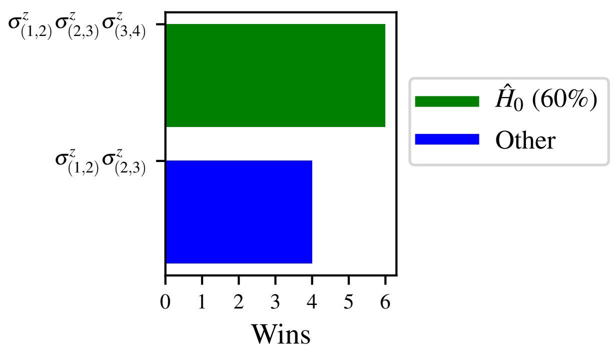
Run level analysis are held in the main results directory and several sub-directories created by the analyse script. Here, we recommend running a number of instances with very few resources so that the test finishes quickly[4]. The results will therefore be meaningless, but allow fo elucidation of the resultant plots. First, reconfigure some settings of Listing C.3 and launch again.

```
num_instances=10
experiments=20
particles=100
run_qhl=0
exploration_strategy=ExampleBasic
```

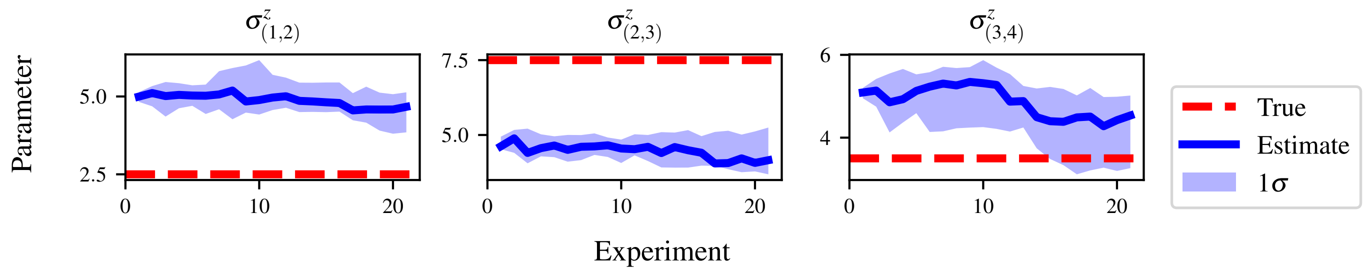
Listing C.13: local_launch configuration for QMLA run.

Some of the generated analysis are shown in Figs. C.4 to C.5. The number of instances for which each model was deemed champion, i.e. their *win rates* are given in Fig. C.4a. The *top models*, i.e. those with highest win rates, analysed further: the average parameter estimation progression for $\hat{H}_0$ – including only the instances where $\hat{H}_0$ was deemed champion – are shown in Fig. C.4b. Irrespecitve of the champion models, the rate with which each term is found in the champion model ($\hat{t} \in \hat{H}'$) indicates the likelihood that the term is really present; these rates – along with the parameter values learned – are shown in Fig. C.4c. The champion model from each instance can attempt to reproduce system dynamics: we group together these reproductions for each model in Fig. C.5.

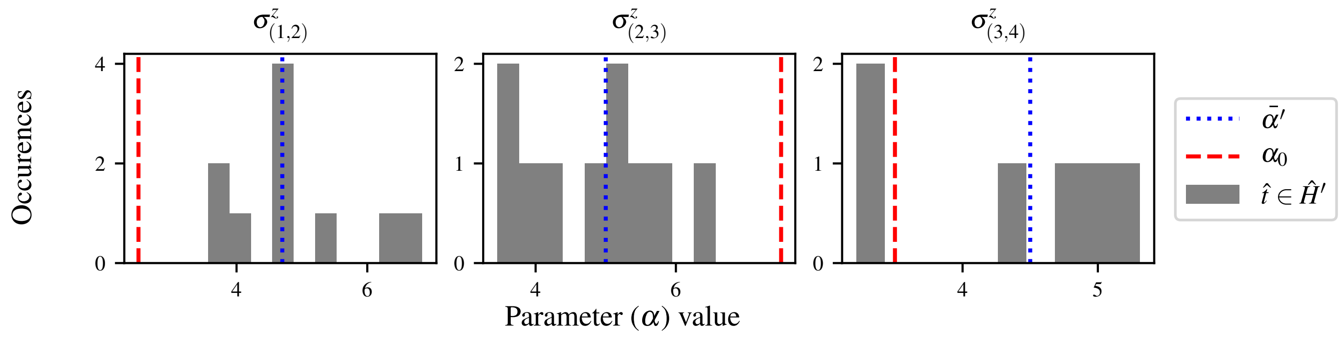
4 This run will take about ten minutes



(a)



(b)



(c)

Figure C.4: QMLA run plots; found within run directory e.g. Jan_01/01_23/. **(a)** performace/model_wins: number of instance wins achieved by each model. **(b)** champion_models/params_params_pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4: parameter estimation progression for the true model , only for the instances where it was deemed champion. **(c)** champion_models/terms_and_params: histogram of parameter values found for each term which appears in any champion model, with the true parameter (α_0) in red and the median learned parameter ($\bar{\alpha}'$) in blue.

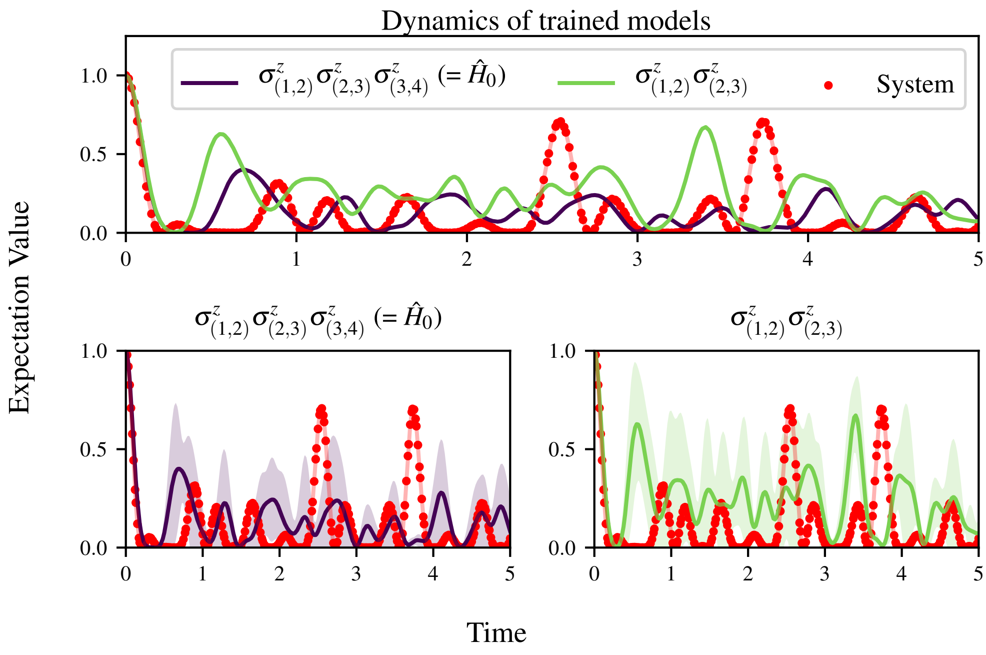


Figure C.5: Run plot performace/dynamics: median dynamics of the champion models. The models which won most instances are shown together in the top panel, and individually in the lower panels. The median dynamics from the models' learnings in its winning instances are shown, with the shaded region indicating the 66% confidence region.

C.3 PARALLEL IMPLEMENTATION

We provide utility to run QMLA on parallel processes. Individual models' training can run in parallel, as well as the calculation of BF between models. The provided script is designed for portable batch system (PBS) job scheduler running on a compute cluster. It will require a few adjustments to match the system being used. Overall, though, it has mostly a similar structure as the local_launch.sh script used above.

QMLA must be downloaded on the compute cluster as in Listing C.1; this can be a new fork of the repository, though it is sensible to test installation locally as described in this chapter so far, then *push* that version, including the new ES, to Github, and cloning the latest version. It is again advisable to create a Python virtual environment in order to isolate QMLA and its dependencies[5]. Open the parallel launch script, QMLA/launch/parallel_launch.sh, and prepare the first few lines as

```
#!/bin/bash
```

5 Indeed it is sensible to do this for any Python development project.

```
##### -------------------------------------------------- #####
# QMLA run configuration
##### -------------------------------------------------- #####
num_instances=10 # number of \glspl{instance} in run
run_qhl=0 # perform QHL on known (true) model
run_qhl_multi_model=0 # perform QHL for defined list of models
experiments=250
particles=1000
plot_level=5


##### -------------------------------------------------- #####
# Choose an exploration strategy
# This will determine how QMLA proceeds.
##### -------------------------------------------------- #####
exploration_strategy="ExampleBasic"
```

Listing C.14: parallel_launch script

When submitting jobs to schedulers like PBS, we must specify the time required, so that it can determine a fair distribution of resources among users. We must therefore *estimate* the time it will take for an instance to complete: clearly this is strongly dependent on the numbers of experiments (N_e) and particles (N_p), and the number of models which must be trained. QMLA attempts to determine a reasonable time to request based on the max_num_models_by_shape attribute of the ES, by calling QMLA/scripts/time_required_calculation.py. In practice, this can be difficult to set perfectly, so the timing_insurance_factor attribute of the ES can be used to correct for heavily over- or under-estimated time requests. Instances are run in parallel, and each instance trains/compares models in parallel. The number of processes to request, N_c for each instance is set as num_processes_to_parallelise_over in the ES. Then, if there are N_r instances in the run, we will be requesting the job scheduler to admit N_r distinct jobs, each requiring N_c processes, for the time specified.

The parallel_launch script works together with launch/run_single_qmla_instance.sh, though note a number of steps in the latter are configured to the cluster and may need to be adapted. In particular, the first command is used to load the redis utility, and later lines are used to initialise a redis server. These commands will probably not work with most machines, so must be configured to achieve those steps.

```

module load tools/redis-4.0.8

...
```

```
SERVER_HOST=$(head -1 "$PBS_NODEFILE")
let REDIS_PORT="6300 + $QMLA_ID"

cd $LIBRARY_DIR
redis-server RedisDatabaseConfig.conf --protected-mode no --port
    $REDIS_PORT &
redis-cli -p $REDIS_PORT flushall
```

Listing C.15: run_single_qmla_instance script

When the modifications are finished, QMLA can be launched in parallel similarly to the local version:

```
source qmla_test/qmla-env/bin/activate

cd qmla_test/QMLA/launch
./parallel_launch.sh
```

Listing C.16: run_single_qmla_instance script

Jobs are likely to queue for some time, depending on the demands on the job scheduler. When all jobs have finished, results are stored as in the local case, in QMLA/launch/results/-Jan_01/01_23, where analyse.sh can be used to generate a series of automatic analyses.

C.4 customising exploration strategys

User interaction with the QMLA codebase should be achieveable primarily through the exploration strategy (ES) framework. Throughout the algorithm(s) available, QMLA calls upon the ES before determining how to proceed. The usual mechanism through which the actions of QMLA are directed, is to set attributes of the ES class: the complete set of influential attributes are available at [22].

QMLA directly uses several methods of the ES class, all of which can be overwritten in the course of customising an ES. Most such methods need not be replaced, however, with the exception of generate_models, which is the most important aspect of any ES: it determines which models are built and tested by QMLA. This method allows the user to impose any logic desired in constructing models; it is called after the completion of every branch of the exploration tree on the ES.

C.4.1 *Greedy search*

A first non-trivial ES is to build models greedily from a set of *primitive* terms, $\mathcal{T} = \{\hat{t}\}$. New models are constructed by combining the previous branch champion with each of the remaining, unused terms. The process is repeated until no terms remain.

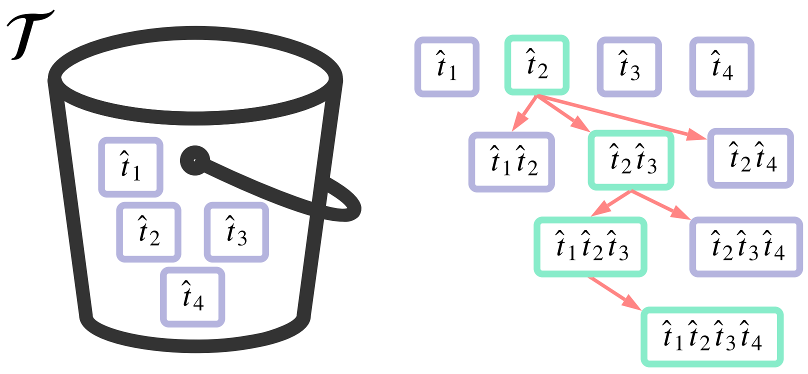


Figure C.6: Greedy search mechanism. **Left**, a set of primitive terms, $\mathcal{T}$, are defined in advance. **Right**, models are constructed from $\mathcal{T}$. On the first branch, the primitve terms alone constitute models. Thereafter, the strongest model (marked in green) from the previous branch is combined with all the unused terms.

We can compose an ES using these rules, say for

$$\mathcal{T} = \left\{ \hat{\sigma}_x^1,\ \hat{\sigma}_y^1,\ \hat{\sigma}_x^1 \otimes \hat{\sigma}_x^2,\ \hat{\sigma}_y^1 \otimes \hat{\sigma}_y^2 \right\}$$

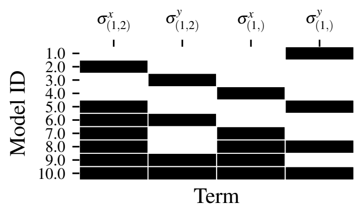
as follows. Note the termination criteria must work in conjunction with the model generation routine. Users can overwrite the method check_tree_completed for custom logic, although a straightforward mechanism is to use the spawn_stage attribute of the ES class: when the final element of this list is Complete, QMLA will terminate the search by default. Also note that the default termination test checks whether the number of branches (spawn_step) exceeds the limit max_spawn_depth, which must be set artifically high to avoid ceasing the search too early, if relying solely on spawn_stage. Here we demonstrate how to impose custom logic to terminate the seach also.

```
class ExampleGreedySearch(
    exploration_strategy.ExplorationStrategy
):
    r"""
    From a fixed set of terms, construct models iteratively,
greedily adding all unused terms to separate models at each
   call to the generate_models.

"""

def __init__(
    self,
    exploration_rules,
    **kwargs
):

    super().__init__(
        exploration_rules=exploration_rules,
        **kwargs
    )
    self.true_model = 'pauliSet_1_x_d3+pauliSet_1J2_yJy_d3+
       pauliSet_1J2J3_zJzJz_d3'
    self.initial_models = None
    self.available_terms = [
        'pauliSet_1_x_d3', 'pauliSet_1_y_d3',
        'pauliSet_1J2_xJx_d3', 'pauliSet_1J2_yJy_d3'
    ]
    self.branch_champions = []
    self.prune_completed_initially = True
    self.check_champion_reducibility = False

def generate_models(
    self,
    model_list,
    **kwargs
):
    self.log_print([
        "Generating models in tiered greedy search at spawn
           step {}.".format(
             self.spawn_step,
        )
    ])
    try:
        previous_branch_champ = model_list[0]
        self.branch_champions.append(previous_branch_champ)
        except:
            previous_branch_champ = ""

        if self.spawn_step == 0 :
            new_models = self.available_terms
        else:
            new_models = greedy_add(
                current_model = previous_branch_champ,
                terms = self.available_terms
            )

        if len(new_models) == 0:
            # Greedy search has exhausted the available models;
            # send back the list of branch champions and
               terminate search.
            new_models = self.branch_champions
            self.spawn_stage.append('Complete')

        return new_models

def greedy_add(
    current_model,
    terms,
):
    r"""
    Combines given model with all terms from a set.

    Determines which terms are not yet present in the model,
    and adds them each separately to the current model.

    :param str current_model: base model
    :param list terms: list of strings of terms which are to be
       added greedily.
    """

    try:
        present_terms = current_model.split('+')
    except:
        present_terms = []
    nonpresent_terms = list(set(terms) - set(present_terms))
```

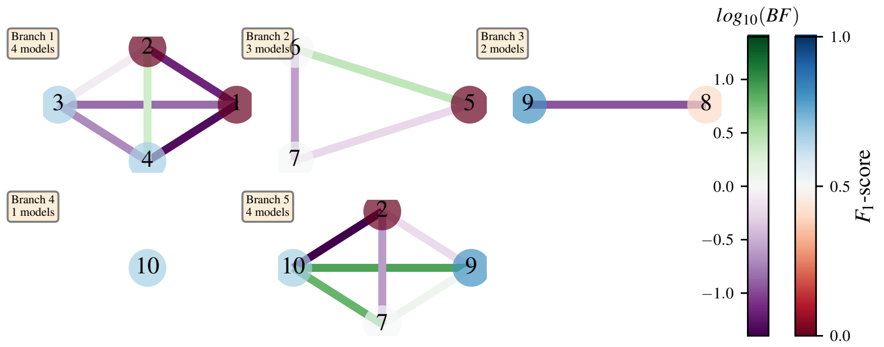
```
        term_sets = [
            present_terms+[t] for t in nonpresent_terms
        ]

        new_models = ["+".join(term_set) for term_set in term_sets]

        return new_models
```

Listing C.17: ExampleGreedySearch exploration stategy

This run can be implemented locally or in parallel as described above[6], and analysed as in Listing C.12, generating figures in accordance with the plot_level set by the user in the launch script. Outputs can again be found in the instances subdirectory, including a map of the models generated, as well as the branches they reside on, and the BFs between candidates, Fig. C.7.



(a)



(b)

Figure C.7: Greedy exploration strategy. **(a)**, composition_of_models. **b**, graphs_of_branches_ExampleGreedySearch: shows which models reside on each branches of the exploration tree. Models are coloured by their F_1-score, and edges represent the BF between models. The first four branches are equivalent to those in Fig. C.6 , while the final branch considers the set of branch champions, in order to determine the overall champion.

6 We advise reducing plot_level to 3 to avoid excessive/slow figure generation.

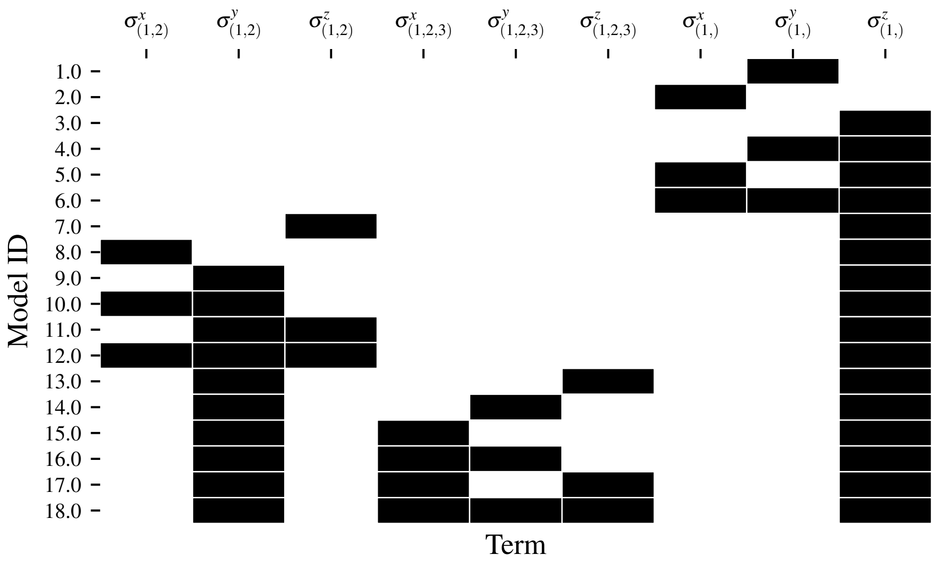
C.4.2 *Tiered greedy search*

We provide one final example of a non-trivial ES: tiered greedy search. Similar to the idea of Appendix C.4.1, except terms are introduced hierarchically: sets of terms $\mathcal{T}_1, \mathcal{T}_2, \dots \mathcal{T}_n$ are each examined greedily, where the overall strongest model of one tier forms the seed model for the subsequent tier. This is depicted in the main text in **??**. A corresponding ES is given as follows.

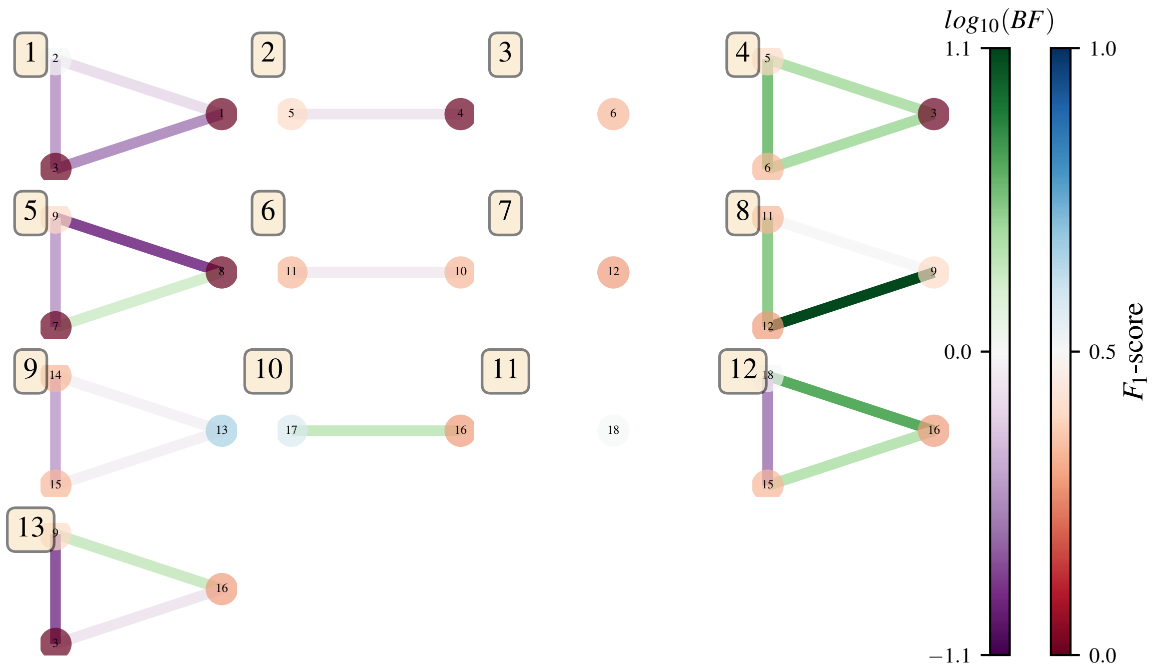
```
class ExampleGreedySearchTiered(
    exploration_strategy.ExplorationStrategy
):
    r"""
    Greedy search in tiers.

    Terms are batched together in tiers;
    tiers are searched greedily;
    a single tier champion is elevated to the subsequent tier.

    """

    def __init__(
        self,
        exploration_rules,
        **kwargs
    ):
        super().__init__(
            exploration_rules=exploration_rules,
            **kwargs
        )
        self.true_model = 'pauliSet_1_x_d3+pauliSet_1J2_yJy_d3+
           pauliSet_1J2J3_zJzJz_d3'
        self.initial_models = None
        self.term_tiers = {
            1 : ['pauliSet_1_x_d3', 'pauliSet_1_y_d3', '
               pauliSet_1_z_d3' ],
            2 : ['pauliSet_1J2_xJx_d3', 'pauliSet_1J2_yJy_d3', '
               pauliSet_1J2_zJz_d3'],
            3 : ['pauliSet_1J2J3_xJxJx_d3', '
               pauliSet_1J2J3_yJyJy_d3', 'pauliSet_1J2J3_zJzJz_d3
               '],
        }
        self.tier = 1
        self.max_tier = max(self.term_tiers)
        self.tier_branch_champs = {k : [] for k in self.
            term_tiers}
        self.tier_champs = {}
        self.prune_completed_initially = True
        self.check_champion_reducibility = True

    def generate_models(
        self,
        model_list,
        **kwargs
    ):
        self.log_print([
            "Generating models in tiered greedy search at spawn
                step {}.".format(
                  self.spawn_step,
            )
        ])

        if self.spawn_stage[-1] is None:
            try:
                previous_branch_champ = model_list[0]
                self.tier_branch_champs[self.tier].append(
                    previous_branch_champ)
            except:
                previous_branch_champ = None

        elif "getting_tier_champ" in self.spawn_stage[-1]:
            previous_branch_champ = model_list[0]
            self.log_print([
                "Tier champ for {} is {}".format(self.tier,
                    model_list[0])
            ])
            self.tier_champs[self.tier] = model_list[0]
            self.tier += 1
            self.log_print(["Tier now = ", self.tier])
            self.spawn_stage.append(None) # normal processing
            if self.tier > self.max_tier:
                self.log_print(["Completed tree for ES"])
                self.spawn_stage.append('Complete')
                return list(self.tier_champs.values())
        else:
            self.log_print([
                "Spawn stage:", self.spawn_stage
            ])

        new_models = greedy_add(
            current_model = previous_branch_champ,
            terms = self.term_tiers[self.tier]
        )
        self.log_print([
            "tiered search new_models=", new_models
        ])

        if len(new_models) == 0:
            # no models left to find - get champions of branches
               from this tier
            new_models = self.tier_branch_champs[self.tier]
            self.log_print([
                "tier champions: {}".format(new_models)
            ])
            self.spawn_stage.append("getting_tier_champ_{}".
               format(self.tier))
        return new_models

    def check_tree_completed(
        self,
        spawn_step,
        **kwargs
    ):
        r"""
        QMLA asks the exploration tree whether it has finished
           growing;
        the exploration tree queries the exploration strategy
           through this method
        """
        if self.tree_completed_initially:
            return True
        elif self.spawn_stage[-1] == "Complete":
            return True
        else:
            return False


def greedy_add(
    current_model,
    terms,
):
    r"""
    Combines given model with all terms from a set.

    Determines which terms are not yet present in the model,
    and adds them each separately to the current model.

    :param str current_model: base model
    :param list terms: list of strings of terms which are to be
       added greedily.
    """

    try:
        present_terms = current_model.split('+')
    except:
        present_terms = []
    nonpresent_terms = list(set(terms) - set(present_terms))

    term_sets = [
        present_terms+[t] for t in nonpresent_terms
    ]

    new_models = ["+".join(term_set) for term_set in term_sets]

    return new_models
```

Listing C.18: ExampleGreedySearchTiered exploration stategy

with corresponding results in Fig. C.8.



(a)



(b)

Figure C.8: Tiered greedy exploration strategy. **(a),** composition_of_models. **b,** graphs_of_branches_ExampleGreedySearchTiered: shows which models reside on each branches of the exploration tree. Models are coloured by their F_1-score, and edges represent the BF between models. In each tier, three branches greedily add terms, and a fourth branch considers the champions of the first three branches in order to nominate a tier champion. The final branch consists only of the tier champions, to nominate the global champion, $\hat{H}'$.

BIBLIOGRAPHY

[1] Christopher E Granade, C Ferrie, N Wiebe, and D G Cory. Robust online Hamiltonian learning. *New Journal of Physics*, 14(10):103013, October 2012.

[2] Nathan Wiebe, Christopher Granade, Christopher Ferrie, and David Cory. Quantum hamiltonian learning using imperfect quantum resources. *Physical Review A*, 89(4):042314, 2014.

[3] N Wiebe, C Granade, C Ferrie, and D G Cory. Hamiltonian Learning and Certification Using Quantum Resources. *Physical Review Letters*, 112(19):190501–5, May 2014.

[4] Antonio A. Gentile, Brian Flynn, Sebastian Knauer, Nathan Wiebe, Stefano Paesani, Christopher E. Granade, John G. Rarity, Raffaele Santagati, and Anthony Laing. Learning models of quantum systems from experiments, 2020.

[5] Jianwei Wang, Stefano Paesani, Raffaele Santagati, Sebastian Knauer, Antonio A Gentile, Nathan Wiebe, Maurangelo Petruzzella, Jeremy L O'Brien, John G Rarity, Anthony Laing, et al. Experimental quantum hamiltonian learning. *Nature Physics*, 13(6):551–555, 2017.

[6] Jane Liu and Mike West. Combined parameter and state estimation in simulation-based filtering. In *Sequential Monte Carlo methods in practice*, pages 197–223. Springer, 2001.

[7] Dominic W Berry, Andrew M Childs, and Robin Kothari. Hamiltonian simulation with nearly optimal dependence on all parameters. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 792–809. IEEE, 2015.

[8] Andrew M Childs, Dmitri Maslov, Yunseong Nam, Neil J Ross, and Yuan Su. Toward the first quantum simulation with quantum speedup. *Proceedings of the National Academy of Sciences*, 115(38):9456–9461, 2018.

[9] Alessandro Rudi, Leonard Wossnig, Carlo Ciliberto, Andrea Rocchetto, Massimiliano Pontil, and Simone Severini. Approximating hamiltonian dynamics with the nyström method. *Quantum*, 4:234, 2020.

[10] Christopher Granade, Christopher Ferrie, Steven Casagrande, Ian Hincks, Michal Kononenko, Thomas Alexander, and Yuval Sanders. QInfer: Library for statistical inference in quantum information, 2016.

[11] Arseni Goussev, Rodolfo A Jalabert, Horacio M Pastawski, and Diego Wisniacki. Loschmidt echo. *arXiv preprint arXiv:1206.6348*, 2012.

[12] Nathan Wiebe, Christopher Granade, and David G Cory. Quantum bootstrapping via compressed quantum hamiltonian learning. *New Journal of Physics*, 17(2):022005, 2015.

[13] Bas Hensen, Hannes Bernien, Anaïs E Dréau, Andreas Reiserer, Norbert Kalb, Machiel S Blok, Just Ruitenberg, Raymond FL Vermeulen, Raymond N Schouten, Carlos Abellán, et al. Loophole-free bell inequality violation using electron spins separated by 1.3 kilometres. *Nature*, 526(7575):682–686, 2015.

[14] Alexandr Sergeevich, Anushya Chandran, Joshua Combes, Stephen D Bartlett, and Howard M Wiseman. Characterization of a qubit hamiltonian using adaptive measurements in a fixed basis. *Physical Review A*, 84(5):052315, 2011.

[15] Christopher Ferrie, Christopher E Granade, and David G Cory. How to best sample a periodic probability distribution, or on the accuracy of hamiltonian finding strategies. *Quantum Information Processing*, 12(1):611–623, 2013.

[16] Christopher E Granade. Characterization, verification and control for large quantum systems. page 92, 2015.

[17] Christopher Ferrie. High posterior density ellipsoids of quantum states. *New Journal of Physics*, 16(2):023006, 2014.

[18] Michael J Todd and E Alper Yıldırım. On khachiyan's algorithm for the computation of minimum-volume enclosing ellipsoids. *Discrete Applied Mathematics*, 155(13):1731–1744, 2007.

[19] Ian Hincks, Thomas Alexander, Michal Kononenko, Benjamin Soloway, and David G Cory. Hamiltonian learning with online bayesian experiment design in practice. *arXiv preprint arXiv:1806.02427*, 2018.

[20] Lukas J Fiderer, Jonas Schuff, and Daniel Braun. Neural-network heuristics for adaptive bayesian quantum estimation. *arXiv preprint arXiv:2003.02183*, 2020.

[21] Brian Flynn. Mathematical introduction to quantum computation, 2015. Undergraduate thesis.

[22] Quantum model learning agent documentation. https://quantum-model-learning-agent.readthedocs.io/en/latest/, Jan 2021. [Online; accessed 12. Jan. 2021].

[23] Brian Flynn. Quantum model learning agent. https://github.com/flynnbr11/QMLA, 2021.