numbers=left,

Doctorate of Philosophy

Schrödinger's Catwalk

Brian Flynn

University of Bristol

November, 2020

CONTENTS

I ALGORITHMS 2

1 QUANTUM HAMILTONIAN LEARNING 3
- 1.1 Sequential Monte Carlo 5
- 1.2 Likelihood 7
  - 1.2.1 Interactive Quantum Likelihood Estimation 7
  - 1.2.2 Analytical likelihood 8
- 1.3 Total log total likelihood 9
- 1.4 Parameter estimation 10
- 1.5 Experiment design heuristic 12
  - 1.5.1 Particle Guess Heuristic 12
  - 1.5.2 Alternative experiment design heuristics 13
- 1.6 Probe selection 15

2 QUANTUM MODEL LEARNING AGENT 16
- 2.1 Models 16
- 2.2 Bayes factors 17
  - 2.2.1 Experiment sets 18
- 2.3 Quantum Model Learning Agent Protocol 19
- 2.4 Exploration Strategies 20
  - 2.4.1 Model generation 21
  - 2.4.2 Decision criteria for the model search phase 23
  - 2.4.3 True model specification 23
  - 2.4.4 Modular functionality 23
  - 2.4.5 Exploration strategy examples 24
- 2.5 Generality 26
  - 2.5.1 Agency 27
- 2.6 Algorithms 27

3 SOFTWARE 33
- 3.1 Implementation 33
  - 3.1.1 Object oriented programming 33
- 3.2 Python framework 36
  - 3.2.1 Application 36
  - 3.2.2 Algorithm 37
  - 3.2.3 Infrastructure 38
  - 3.2.4 Analyses and plots 39
- 3.3 Usage 39

ACRONYMS

BF Bayes factor. 17–19, 25–27

CLE classical likelihood estimation. 7

EDH experiment design heuristic. 12–15, 18, 27
ES exploration strategy. 19–27
ET exploration tree. 20, 21, 23, 24, 26, 27

GA genetic algorithm. 26

HPD high particle density. 11

IQLE interactive quantum likelihood estimation. 7, 8

LTL log total likelihood. 9

ML machine learning. 18, 19
MVEE minimum volume enclosing ellipsoid. 11

NV nitrogen-vacancy. 3
NVC nitrogen-vacancy centre. 8

PGH particle guess heuristic. 12–15

QHL quantum Hamiltonian learning. 3–8, 10, 12, 13, 15, 17–19, 23, 24, 26, 27
QL quadratic loss. 10
QLE quantum likelihood estimation. 7
QMLA Quantum Model Learning Agent. 7, 16, 17, 19–23, 26, 27

SMC sequential monte carlo. 5–7, 9, 12–14, 23

TLTL total log total likelihood. 10, 17–19, 26

GLOSSARY

Loschmidt echo (LE) Quantum chaotic effect described. . 7, 8
model search (MS) A model search . 19–21, 23, 27

hyperparameter Variable within an algorithm that determines how the algorithm itself proceeds.. 5

likelihood Value that represents how likely a hypothesis is.. 4, 7, 9, 12, 23, 24, 26, 28

model The mathematical description of some quantum system. 16

probe Input probe state, $|\psi\rangle$, which the target system is initialised to, before unitary evolution. plural. 7, 8, 12, 15

spawn Process by which new models are generated by combining previously considered models.. 21

term Individual constituent of a model, e.g. a single operator within a sum of operators, which in total describe a Hamiltonian. . 16

volume Volume of a parameter distribution's credible region.. 10, 11

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 apriori. 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 $d \in \{0, 1\}$, 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, 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 consistenst with those measured from Q.

The procedure is as follows. A *prior* probability distribution $\Pr(\vec{\alpha})$ 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. An example of imposing such domain-specific prior knowledge is, when choosing the prior for a model representing an e^- spin in a nitrogen-vacancy (NV) centre, to select GHz parameters for the electron spin's rotation terms, and MHz terms for the spin's coupling to nuclei, as proposed in literature. 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

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

(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 cost of running QHL for a single 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.

Bayes Rule

Bayes' rule is used to update a probability distribution describing hypotheses, Pr(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(hypothesis), multiplied by the likelihood that the new data would be observed were that hypothesis true, Pr(data|hypothesis), normalsied by the probability of observing that data in the first place, Pr(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 then discretise Eq. (1.2) to 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 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 hypothesis.

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, 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 1.2). It is necessary first to know the datum d (either 0 or 1) which was projected by Q under real 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 – i.e. we compute a *posterior* probability distribution by performing this routine on N_p particles.

SEQUENTIAL MONTE CARLO

In practice, QHL samples from and updates $\Pr(\vec{\alpha})$ via 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 are updated by Eq. (1.5).

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

In this way, strong particles (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 thus: we *simulataneously* update sampled particles' weights as well as $\Pr(\vec{\alpha})$. This iterates for the following 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 $\Pr(\vec{\alpha})$, according to the Liu-West resampling algorithm [6]. 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 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

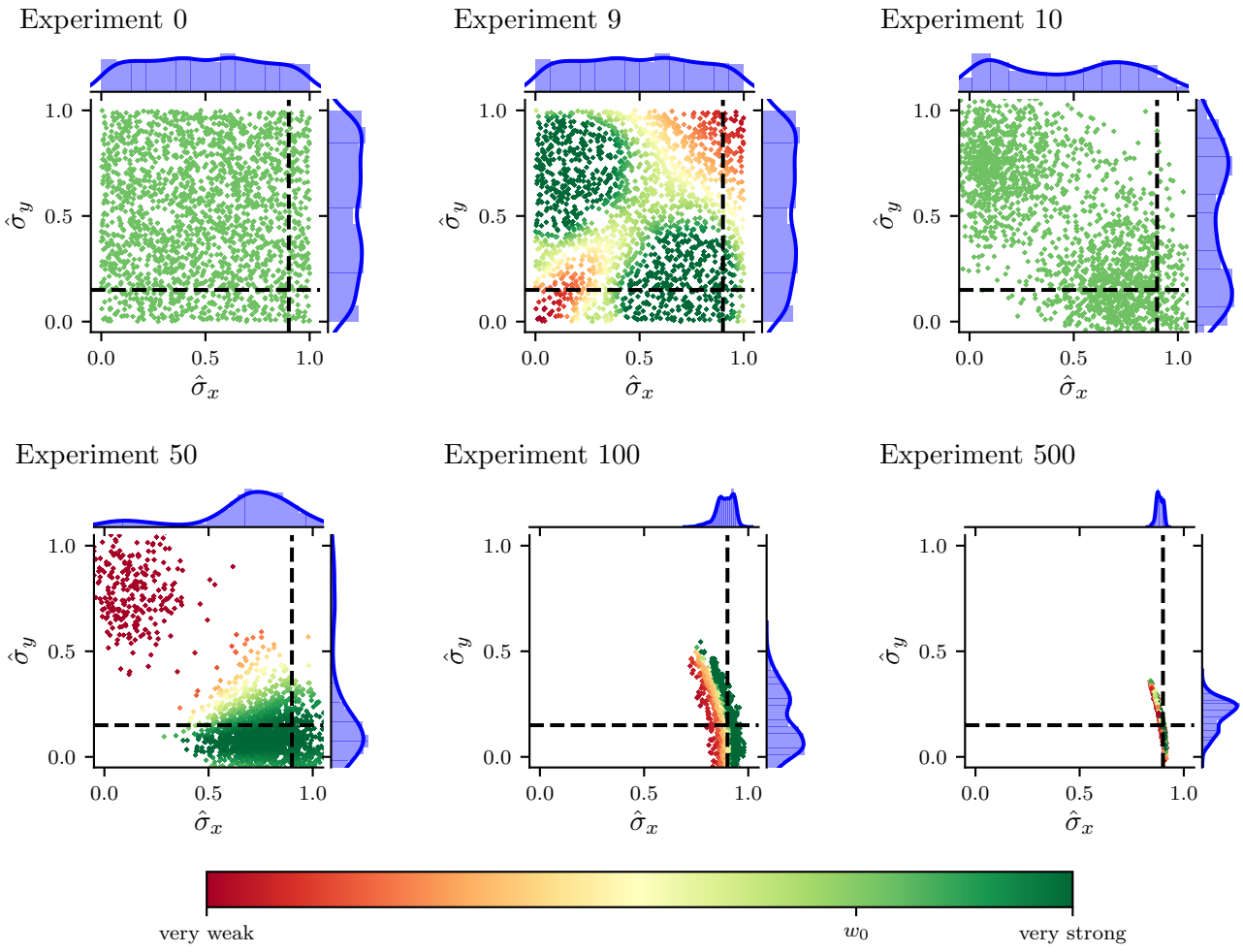

Figure 1.1: Quantum Hamiltonian learning (QHL) via sequential monte carlo (SMC). The studied model has two terms, $\{\hat{\sigma}^x, \hat{\sigma}^y\}$ with true parameters $\alpha_x = 0.9, \alpha_y = 0.15$ (dahsed lines), with $N_e = 500, N_p = 2000$. Crosses represent particles, while the distribution $\Pr(\alpha_p)$ 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 sufficintly 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.

presented in Fig. 1.1, where a two-parameter Hamiltonian is learned starting from a uniform distribution.

LIKELIHOOD

The fundamental step within QHL is the calculation of likelihood in Eq. (1.3). The core of this learning algorithm is that this likelihood can be retrieved from the Born rule, although in principle *any* valid likelihood function can fulfil this equation, provided the calculation of the likelihood captures the probabilty that the present hypothesis produced the present datum.

In general, it is not always possible to derive the analytical likelihood, especially in cases where we wish to vary the probe. When Eq. (1.4) can be computed classically, QHL relies on classical likelihood estimation (CLE), i.e. involving the exponential calculation of Eq. (1.4), whereas quantum likelihood estimation (QLE) uses the same likelihood function computed on a quantum simulator; this is the sole application of quantum simulators in this protocol and indeed the remainder of this thesis. Access to such hardware, operating perfectly, would provide exponential speedup in the calculation of this term, rendering both QHL and the wider Quantum Model Learning Agent (QMLA) formalism scalable, although in this thesis we do not implement QLE so everything can be viewed as CLE. QLE was implemented in [4].

Interactive Quantum Likelihood Estimation

An important extension to QLE is interactive quantum likelihood estimation (IQLE), which follows SMC but uses an alternative likelihood function in order to overcome some of its inherent challenges [3]. Two almost identical Hamiltonians will diverge after exponentially small evolution time [7]. This is problematic for QHL because it relies on the likelihood function which is built on the assumption that increasing accuracy of $\hat{H}(\vec{\alpha})$ approximating $\hat{H}_0$ should result in rising likelihood; this result indicates that even extremely accurate approximations become unreliable after very short evolution times. Coupled with the result that small time experiments are uninformative [8], this observation demands exponentially many measurements to approximate the exponentially small likelihoods, rendering the approach inefficient.

The Loschmidt echo (LE) is the measure of the revival resulting from an imperfect time-reversal operation implemented after the standard time evolution. The reversal operation corresponds to some Hamiltonian, $\hat{H}_-$, which is seen as an attempt to un-do the evolution according to the original Hamiltonian, $\hat{H}_+$. As such we say that $\hat{H}_-$ is evolved for $-t$, so its unitary is $e^{-i\hat{H}_-(-t)}$ after $\hat{H}_+$ is evolved for t. The LE can be written and characterised as

$$M(t) = \left|\langle\psi|e^{+i\hat{H}_- t}e^{-i\hat{H}_+ t}|\psi\rangle\right|^2 \sim \begin{cases} 1-\mathcal{O}(t^2), & t \leq t_c \\ e^{-\mathcal{O}(t)}, & t_c \leq t \leq t_s \\ 1/\|\hat{H}\|, & t \geq t_s \end{cases} \tag{1.6}$$

where $\hat{H}_-, \hat{H}_+$ are backward and forward time evolutions respsectively, which are assumed almost identical; $\|\hat{H}\|$ is their dimension, 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 [9]. In effect, the LE gaurantees that 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 $\hat{H}_0$ (true) or $\hat{H}(\vec{\alpha})$ (particle or hypothesis), and sampling $\hat{H}_-$ from $\Pr(\vec{\alpha})$, we can adopt Eq. (1.6) as the likelihood function in Eq. (1.4), in the knowledge that this will disinguish between hypotheses based on their similarity to $\hat{H}_0$.

Importantly, IQLE can only be used where we can *reliably* evolve 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 [10]. The remaining application for IQLE, and correspondingly QHL, is in the characterisation of untrusted quantum simulators, which can realise such coherent channels [4].

Analytical likelihood

In some cases, analytical likelihood functions can be derived to describe the dynamics of simple quantum systems [11, 12], for instance encoding the Rabi frequency ω of an oscillating electron spin in an nitrogen-vacancy centre (NVC),

$$\hat{H}(\omega) = \frac{\omega}{2}\hat{\sigma}_z. \tag{1.7}$$

Then, bearing in mind 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. (1.7) is given by

$$\begin{aligned} U = e^{-i\hat{H}(\omega)t} = e^{-i\frac{\omega t}{2}\hat{\sigma}_z} &= \cos\left(\frac{\omega t\hat{\sigma}_z}{2}\right) - i\sin\left(\frac{\omega t\hat{\sigma}_z}{2}\right) \\ &= \left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k)!}\left(\frac{\omega t}{2}\right)^{2k}\hat{\sigma}_z^{2k}\right) - i\left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)!}\left(\frac{\omega t}{2}\right)^{2k+1}\hat{\sigma}_z^{2k+1}\right) \\ &= \left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k)!}\left(\frac{\omega t}{2}\right)^{2k+1}\right)\hat{\mathbb{1}} - i\left(\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)!}\left(\frac{\omega t}{2}\right)^{2k+1}\right)\hat{\sigma}_z \\ &= \cos\left(\frac{\omega t}{2}\right)\hat{\mathbb{1}} - i\sin\left(\frac{\omega t}{2}\right)\hat{\sigma}_z \end{aligned} \tag{1.8}$$

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

$$\langle\psi_1|\,U\,|\psi_0\rangle = \cos\left(\frac{\omega t}{2}\right)\langle\psi_1|\psi_0\rangle - i\sin\left(\frac{\omega t}{2}\right)\langle\psi_1|\hat{\sigma}_z|\psi_0\rangle\,. \tag{1.9}$$

By initialising and projecting into the same state, say $|\psi_0\rangle = |\psi_1\rangle = |+\rangle$, we have

$$\begin{aligned} \hat{\sigma}_z |+\rangle = |-\rangle \implies \langle\psi_1|\hat{\sigma}_z|\psi_0\rangle &= 0 \\ \langle\psi_1|\psi_0\rangle &= 1 \\ \implies \langle\psi_1|U|\psi_0\rangle &= \cos\left(\frac{\omega t}{2}\right), \end{aligned} \tag{1.10}$$

i.e. if the system measures in $|+\rangle$, we set the datum $d = 1$, otherwise $d = 0$. Then, 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,

$$\Pr(d = 1|\omega; t) = |\langle\psi_1|U|\psi_0\rangle|^2 = \cos^2\left(\frac{\omega t}{2}\right) \tag{1.11}$$

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

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*,

$$l_e = \sum_{p\in\{p\}} \Pr(d|\vec{\alpha}_p; e) \times w_p^{old}. \tag{1.12}$$

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), [13].

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

$$\begin{aligned} w_p^0 = \frac{1}{N_p} &\implies \sum_p^{N_p} w_p^0 = 1; \\ \Pr(d|\vec{\alpha}_p; e) \leq 1 &\implies \Pr(d|\vec{\alpha}_p; e) \times w_p^{old} \leq w_p^{old} \\ &\implies \sum_{\{p\}} \Pr(d|\vec{\alpha}_p; e) \times w_p^{old} \leq \sum_{\{p\}} w_p^{old} \leq \sum_p^{N_p} w_p^0; \\ &\implies l_e \leq 1. \end{aligned} \tag{1.13}$$

Eq. (1.12) 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),

$$\mathcal{L}_i = \sum_{e \in \mathcal{E}} ln(l_e). \tag{1.14}$$

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, i.e. $\hat{H}_i$ 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 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 poor agreement $l_e < 1 \Rightarrow ln(l_e) < 0$. Then, in all cases Eq. (1.14) is negative, and across a series of experiments, strong agreement gives low $|\mathcal{L}_i|$, whereas weak agreement gives large $|\mathcal{L}_i|$.

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. Here we use the quadratic loss (QL), which captures this error through the sum of the square difference between each parameter's true and estimated values symetrically, i.e. error above the true parameter is as impactful as error below. We can record the QL at each experiment of our training regime and hence track its performance.

Definition 1.4.1 (Quadratic Loss)**.** For a true parameterisation $\vec{\alpha}_0$, and a hypothesis $\vec{\alpha}$, the quadratic loss is given by

$$L_Q = \|\vec{\alpha}_0 - \vec{\alpha}\|^2. \tag{1.15}$$

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 $\text{mean}\left(\Pr(\vec{\alpha})\right)$ is accurate. For example, for a single parameter ω, our best knowledge of the

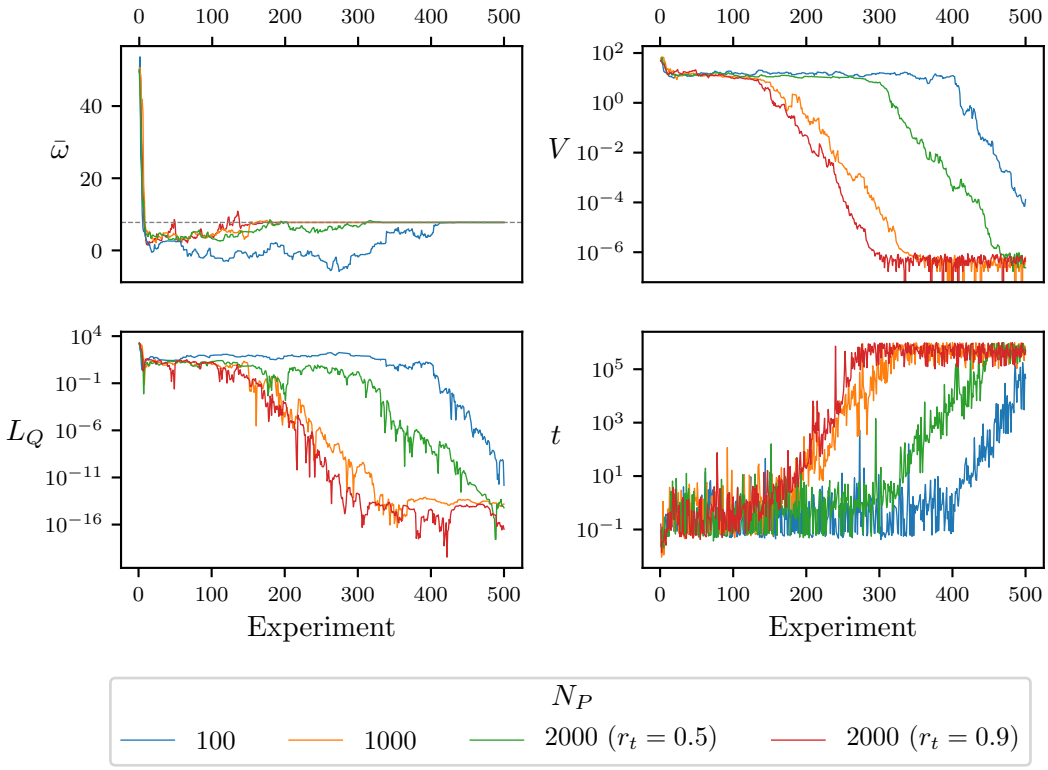

Figure 1.2: Parameter learning for the analyitcal likelihood Eq. (1.11) for varying numbers of particles N_p, for $N_e = 500$. For $N_p = 2000$, we show the resampler threshold set to $r = 0.5$ and $r = 0.9$. (a) the parameter estimate, i.e. $\bar{\omega}$, the mean of the posterior distribution after 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.

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 [14, 15].

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 $\sum_{p \in \mathcal{P}} w_p \geq \kappa$. We use the concept minimum volume enclosing ellipsoid (MVEE) to capture the confidence region [15], calculated as in [16], which are characterised by their covariance matrix, Σ, which allows us to calculate the volume,

do we use MVEE or the posterior directly as the credible region?

do we use MV posterior direc credible regio

$$V(\Sigma) = \frac{\pi^{|\vec{\alpha}|/2}}{\Gamma(1 + \frac{|\vec{\alpha}|}{2})} \det\left(\Sigma^{-\frac{1}{2}}\right), \tag{1.16}$$

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 cau-

tious 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 fast or exponentially decreasing volume indicates that the parameters are being learned well. When the volume has converged, the learning has saturated and there is little benefit to running further experiments.

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. Usually, the EDH attempts to exploit the information available, adaptively accounting for some aspects of the inference process performed already, although there may be justification for enforcing a non-adaptive schedule, for instance to force QHL to train on a full set of experimental data rather than a restricted set which adaptive methods would advise. We can categorise each EDH as either *online* of *offlnie*, 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 [17, 18].

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. 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 $\hat{H}_0$. For example, consider Eq. (1.7) with a single parameter with $\omega_0 = 10$, and current mean $(\Pr(\omega)) = 9, \text{std}(\Pr(\omega)) = 2$, we can expect that the approximation $\omega' = \text{mean}(\Pr(\omega))$ is valid up to $t_{max} = 1/\text{std}(\Pr(\omega))$. 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 the posterior can comfortably explain, so it will expose the relative difference in likelihood between the posterior's particles, providing a strong capacity to learn. Informally, as the uncertainty in the posterior shrinks, PGH selects

larger times to ensure the training is based on informative eperiments simulataneously with 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 $t_k = {}^{1.26}/_{\text{std}(\Pr(\omega))}$ [12].

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

$$t_k = \frac{1}{\|\vec{\alpha}_i - \vec{\alpha}_j\|}, \tag{1.17}$$

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 $\Pr(())\vec{\alpha}$.

We adopt PGH as the default EDH throughout this thesis, but will have cause to deviate in paricular circumstances.

Alternative experiment design heuristics

We briefly summarise some alternative EDHs which we will use later in this thesis.

Forced exponential increase

One early candidate for a viable EDH was

$$t_k = \left(\frac{9}{8}\right)^k, \tag{1.18}$$

where k is the experiment number. This forces QHL to consider exponentially-increasing times, irrespective of its current state of knowledge.

Fixed set

By providing a list of times, chosen in advance, we can attempt to ensure that QHL tries to account for an entire dataset. For instance, if only a small set of experimental measurements are available, it is sensible to train on all of them, perhaps repeatedly.

Random time upper bounded

Given t_{max}, time is chosen randomly: $t_k = \mathcal{U}(0, t_{max})$.

Mixed: PGH and fixed set

In cases where we wish to allow SMC to train both adaptively and according to a fixed schedule, we can employ this EDH. For instance, where a small set of experimental data is available

offline, we can first use the adapative routine (PGH) to coarsely refine the distribution, but if the parameters cannot be learned sufficiently, e.g. because of noise in the data, the posterior will never be narrow enough to allow the PGH to reach higher experimentally measured times. In this case, it is reasonable to force the EDH to focus some of the learning resources on the data for higher times, especially where much of the underlying physics of interest only starts to be visible for those times, e.g. where the experimental system starts to decohere.

Altered PGH

A straightforward alternative is simply to replace the reciprocal in Eq. (1.17) with any another distance metric, $d(\vec{\alpha}_i, \vec{\alpha}_j)$. In doing so we retain much of the reasoning for PGH. While PGH uses the Euclidean distance, we can instead consider, for example, any of the distance metrics available in `scipy.spatial.distance.pdist` [19], of which in particular we implement `cityblock`, `euclidean`, `chebyshev`, `canberra`, `braycurtis`, `minkowski`.

Volume adaptive PGH

Where it is suspected that PGH is performing poorly, it is feasible to force the EDH to consider orders of magnitude greater or smaller than the distance $d(\vec{\alpha}_i, \vec{\alpha}_j)$. We include a heuristic which tracks the change in volume across experiments, given by $\Delta V = 1 - {}^{V'}/_{V}$, where V is the volume of the prior and V' is the volume of the posterior, noting that several experiments may elapse between measuring V, V', such that we can interpret

$$\begin{cases} V = V' \,(\text{no change}) & \Delta V = 0 \\ V' > V \,(\text{disimprovement}) & \Delta V < 0 \\ V' < V \,(\text{improvement}) & 0 < \Delta V < 1 \end{cases} \tag{1.19}$$

Eq. (1.19) allows us to set acceptable levels for the learning via the volume reduction: when the change in volume is deemed inappropriate, we force the experimental times into another order of magnitude, by imposing a scalar factor on the PGH chosen time, again facilitating any distance metric, d,

$$t_k = \kappa_k \frac{1}{d(\vec{\alpha}_i, \vec{\alpha}_j)}, \tag{1.20}$$

where κ_k is a scalar set for each experiment k, initially set as $\kappa_0 = 1$. For example, if $\Delta V > 0.2$ across a small number of experiments (say 10), we see that the volume has reduced by 20% over 10 experiments, which is a strong rate of learning, so we retain $\kappa_k = \kappa_{k-1}$. Conversely, $\Delta V = 0.01$ indicates that the learning has stalled, to compensate, we set $\kappa_k = 10\kappa_{k-1}$, such that SMC challenges $\Pr(\vec{\alpha})$ to explain much more challenging experiments. Likewise, in the case where $\Pr(\vec{\alpha})$ disimproves significantly, say $\Delta V < -0.2$, we can force the heuristic to design conservative experiments by setting $\kappa_k = {}^{\kappa_{k-1}}/_{10}$.

Sampled order of magnitude

When a parameter distribution contains parameters of varying magnitude, it is challenging to select a time that suitably trains all parameters simulataneously. One attempt to resolve this problem is to treat the parameters independently, effectively trying to train them separately. We sample from the individual uncertainty on each parameter, approximated as the square root of the covariance matrix's diagnoal element corresponding to that parameter. For example, suppose a three parameter distribution involves kHz, MHz and GHz terms: the diagonal of the covariance matrix might read $(10^8, 10^{27}, 10^{85})$, such that the individual uncertainties are $(10^{2.8}, 10^9, 10^{9.2})$. Recognising that parameters of higher order of magnitude have a greater impact on dynamics, we assign sampling probability of focusing on that parameter scale according to log_{10} of the individual uncertainties. Then, the probabilities are $\{\frac{2.8}{21}, \frac{9}{21}, \frac{9.2}{21}\}$ for the kHz, MHz, GHz terms respectively. Then, when a parameter scale is chosen, we take the distance between two sampled particles $\vec{\alpha}_i, \vec{\alpha}_j$ (as in PGH), only with respect to that scale, e.g. if MHz is sampled, $t_k \sim \mathcal{O}(10^{-6}s)$.

PROBE SELECTION

A final consideration for 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 [4, 12]. 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, $\Psi = \{|\psi\rangle\}$, of size $N_\psi = |\Psi|$, from which a probe is chosen at each experiment, allowing for the same $|\psi\rangle$ to be used in training of multiple experiments. Ψ can be generated with respect to the individual learning problem as we will examine later, although some straightforward strategies for generating Ψ are

- $|\psi\rangle$ are random, separable probes;
- Ψ is the tomographic basis set;
- $\Psi = \{|+\rangle\}, \quad N_\psi = 1.$

2

QUANTUM MODEL LEARNING AGENT

Quantum Model Learning Agent (QMLA) is an algorithm that extends the concept of applying machine learning to the characterisation of Hamiltonians we've seen so far. The extension, and central question of QMLA is: if we do not even have the structure of the model which describes a target quantum system, can we still learn about the physics of the system? That is, we remove the assumption about the form of the Hamiltonian model, and attempt to uncover which *terms* constitute the Hamiltonian, and in so doing, learn what interactions the system is subject to. Throughout this thesis, we are concerned solely with Hamiltonian models. although, in general, the description of a quantum system of interest need not be Hamiltonian, e.g. it could be Lindbladian for open quantum systems, so we generalise the effort to the search for the most suitable *model*.

For the remainder of this thesis, our objective is to learn the model underlying some target system Q. We will first introduce some concepts which will prove useful when discussing QMLA, before describing the protocol in detail in Section 2.3.

MODELS

Models are simply the mathematical objects which can be used to predict the behaviour of a system. In this thesis, models are synonymous with Hamiltonians, composed of a set of terms, $\mathcal{T} = \{\hat{t}\}$, where each $\hat{t}$ is a matrix. Each term is assosiated with a multipicative scalar, which may be referred to as that term's *parameter*: we impose order on the terms and parameters such that we can succinctly summarise any model as

$$\hat{H} = \begin{pmatrix} \alpha_0 & \dots & \alpha_n \end{pmatrix} \begin{pmatrix} \hat{t}_1 \\ \vdots \\ \hat{t}_n \end{pmatrix} = \vec{\alpha} \cdot \vec{T} \tag{2.1}$$

where $\vec{\alpha}, \vec{T}$ are the model's parameters and terms, respectively.

For example, a model which is the sum of the (non-identity) Pauli operators is given by

$$\begin{aligned} \hat{H} &= \begin{pmatrix} \alpha_x & \alpha_y & \alpha_z \end{pmatrix} \cdot \begin{pmatrix} \hat{\sigma}^x \\ \hat{\sigma}^y \\ \hat{\sigma}^z \end{pmatrix} \\ &= \alpha_x \hat{\sigma}^x + \alpha_y \hat{\sigma}^y + \alpha_z \hat{\sigma}^z \\ &= \begin{pmatrix} \alpha_z & \alpha_x - i\alpha_y \\ \alpha_x + i\alpha_y & \alpha_z \end{pmatrix}. \end{aligned} \tag{2.2}$$

Through this formalism, we can say that the sole task of QHL was to optimise $\vec{\alpha}$, given $\vec{T}$. The principle task of QMLA is to identify $\vec{T}$ with the most statistical evidence for describing the target system Q. In short, QMLA proposes candidate models $\hat{H}_i$ as hypoteses to explain Q; we *train* each model independently through a parameter learning routine, and finally nominate the model with the best performance after training. In particular, QMLA uses QHL as the parameter learning subroutine, but in principle this step can be performed by any algorithm which learns $\vec{\alpha}$ for given $\vec{T}$, such as tomography [20]. While discussing a model $\hat{H}_i$, their *training* then simply means the implementation of QHL, where $\hat{H}_i$ is assumed to represent Q, such that $\vec{\alpha}_i$ is optimised as well as it can be, even if $\hat{H}_i$ is entirely inaccurate.

BAYES FACTORS

We can use the tools introduced in Section 1.3 to *compare* models. Of course it is first necessary to ensure that each model has been adequately trained: while inaccurate models are unlikely to strongly capture the system dynamics, they should first train on the system to determine their best attempt at doing so, i.e. they should undergo the process in Chapter 1. It is statistically meaningful to compare models via their TLTL, $\mathcal{L}_i$, if and only if they have considered the same data, i.e. if models have each attempted to account for the same set of experiments, $\mathcal{E}$.

We can then exploit direct pairwise comparisons between models, by imposing that both models' TLTL are computed based on *any* shared set of experiments $\mathcal{E}$, with corresponding measurements $\mathcal{D} = \{d_e\}_{e\in\mathcal{E}}$. Pairwise comparisons can then be quantified by the Bayes factor (BF),

$$B_{ij} = \frac{\Pr(\mathcal{D}|\hat{H}_i;\mathcal{E})}{\Pr(\mathcal{D}|\hat{H}_j;\mathcal{E})}. \tag{2.3}$$

Intuitively, we see that the BF is the ratio of the likelihood, i.e. the performance, of model $\hat{H}_i$'s attempt to account for the data set $\mathcal{D}$ observed following the experiment set $\mathcal{E}$, against the same likelihood for model $\hat{H}_j$. BFs are known to be statistically signicative of the stronger model from a pair, at explaining observed data, while favouring models of low cardinality, thereby supressing overfitting models.

We have that, for independent experiments, and recalling Eq. (1.12),

$$\begin{aligned}\Pr(\mathcal{D}|\hat{H}_i;\mathcal{E}) &= \Pr(d_n|\hat{H}_i;e_n)\times\Pr(d_{n-1}|\hat{H}_i;e_{n-1})\times\cdots\times\Pr(d_0|\hat{H}_i;e_0)\\ &= \prod_{e\in\mathcal{E}}\Pr(d_e|\hat{H}_i;e)\\ &= \prod_{e\in\mathcal{E}}(l_e)_i.\end{aligned} \tag{2.4}$$

We also have, from Eq. (1.14)

$$\begin{aligned} \mathcal{L}_i &= \sum_{e\in\mathcal{E}} ln\left((l_e)_i\right) \\ \implies e^{\mathcal{L}_i} &= \exp\left(\sum_{e\in\mathcal{E}} ln\left[(l_e)_i\right]\right) = \prod_{e\in\mathcal{E}} \exp\left(ln\left[(l_e)_i\right]\right) = \prod_{e\in\mathcal{E}} (l_e)_i. \end{aligned} \tag{2.5}$$

So we can write

$$B_{ij} = \frac{\Pr(\mathcal{D}|\hat{H}_i;\mathcal{E})}{\Pr(\mathcal{D}|\hat{H}_j;\mathcal{E})} = \frac{\prod_{e\in\mathcal{E}}(l_e)_i}{\prod_{e\in\mathcal{E}}(l_e)_j} = \frac{e^{\mathcal{L}_i}}{e^{\mathcal{L}_j}} \tag{2.6}$$

$$\implies B_{ij} = e^{\mathcal{L}_i - \mathcal{L}_j} \tag{2.7}$$

This is simply the exponential of the difference between two models' total log-likelihoods when presented the same set of experiments. Intuitively, if $\hat{H}_i$ performs well, and therefore has a high TLTL, $\mathcal{L}_i = -10$, and $\hat{H}_j$ performs worse with $\mathcal{L}_j = -100$, then $B_{ij} = e^{-10-(-100)} = e^{90} \gg 1$. Conversely for $\mathcal{L}_i = -100, \mathcal{L}_j = -10$, then $B_{ij} = e^{-90} \ll 1$. Therfore $\left|B_{ij}\right|$ is the strength of the statistical evidence in favour of the interpretation

$$\begin{cases} B_{ij} > 1 & \Rightarrow \hat{H}_i \text{ stronger than } \hat{H}_j \\ B_{ij} < 1 & \Rightarrow \hat{H}_j \text{ stronger than } \hat{H}_i \\ B_{ij} = 1 & \Rightarrow \hat{H}_i \text{ as strong as } \hat{H}_j \end{cases} \tag{2.8}$$

Experiment sets

As mentioned it is necessary for the TLTL of both models in a BF calculation to refer to the same set of experiments, $\mathcal{E}$. There are a number of ways to achieve this, which we briefly summarise here for reference later.

During training (the QHL subroutine), candidate model $\hat{H}_i$ is trained against $\mathcal{E}_i$, designed by an EDH to optimise parameter learning specifically for $\hat{H}_i$; likewise $\hat{H}_j$ is trained on $\mathcal{E}_j$. The simplest method to compute the BF is to enforce $\mathcal{E} = \mathcal{E}_i \cup \mathcal{E}_j$ in Eq. (2.3), i.e. to cross-train $\hat{H}_i$ using the data designed specifically for training $\hat{H}_j$, and vice versa. This is a valid approach because it challenges each model to attempt to explain experiments designed explicitly for its competitor, at which only truly accurate models are likely to succeed.

A second approach builds on the first, but incorporates *burn–in* time in the training regime: this is a standard technique in the evaluation of machine learning (ML) models whereby its earliest iterations are discounted for evaluation so as not to skew its metrics, ensuring the evaluation reflects the strength of the model. In BF, we achieve this by basing the TLTL only on a subset of the training experiments. For example, the latter half of experiments designed

during the training of $\hat{H}_i$, $\mathcal{E}_i'$. This does not result in less predictive BF, since we are merely removing the noisy segments of the training for each model, e.g. the first half of experiments in Fig. 1.2. Moreover it provides a benefit in reducing the computation requirements: updating each model to ensure the TLTL is based on $\mathcal{E}' = \mathcal{E}_i' \cup \mathcal{E}_j'$ requires only half the computation time, which can be further reduced by lowering the number of particles used during the update, N_p', which will give a similar result as using N_p, assuming the posterior has converged.

figures to prove these points.

figures to prove these points.

A final option is to design a set of *evaluation* experiments, $\mathcal{E}_v$, that are valid for a broad variety of models, and so will not favour any particular model. Again, this is a common technique in ML: to use one set of data for training models, and a second, unseen dataset for evaluation. This is clearly a favourable approach: provided for each model we compute Eq. (1.14) using $\mathcal{E}_v$, we can automatically select the strongest model based solely on their TLTLs, meaning we do not have to perform further computationally-expensive updates, as required to cross-train on opponents' experiments during BF calculation. However, it does impose on the user to design a *fair* $\mathcal{E}_v$, requiring unbiased probe states $\{|\psi\rangle\}$ and times $\{t\}$ on a timescale which is meaningful to the system under consideration. For example, experiments with $t > T_2$, the decoherence time of the system, would result in measurements which offer little information, and hence it would be difficult to extract evidence in favour of any model from experiments in this domain. It is difficult to know, or even estimate, such meaningful time scales a priori, so it is difficult for a user to design $\mathcal{E}_v$. Additionally, the training regime each model undergoes during QHL is designed to provide adaptive experiments that take into account the specific model entertained, to choose an optimal set of evolution times, so it is likely that the set of times in $\mathcal{E}_i$ is *reasonable* by default. This approach would be favoured in principle, in the case where such constraints can be accounted for, e.g. an experiment repeated in a laboratory where the available probe states are limited and the timescale achievable is understood.

QUANTUM MODEL LEARNING AGENT PROTOCOL

Given a target quantum system, Q, described by some *true* model $\hat{H}_0$, QMLA distills a model $\hat{H}' \approx \hat{H}_0$. We can think of QMLA as a forest search algorithm[1]: consisting of a number of trees, each of which can have an arbitrary number of branches, where each leaf on each branch is an individual model, QMLA is the search for the leaf in the forest with the strongest statistical evidence of representing Q. Each tree in the QMLA forest corresponds to an independent *model search (MS)*, structured according to a bespoke exploration strategy (ES), which we detail in Section 2.4. In short, the MS proceeds by grouping models in *layers*, training each model on each layer independently, layers are then *consolidated* to rank their performance, and new models are then *spawned*.

1 Note QMLA is not a random forest, where decision trees are added at random, because in QMLA trees are highly structured and included manually.

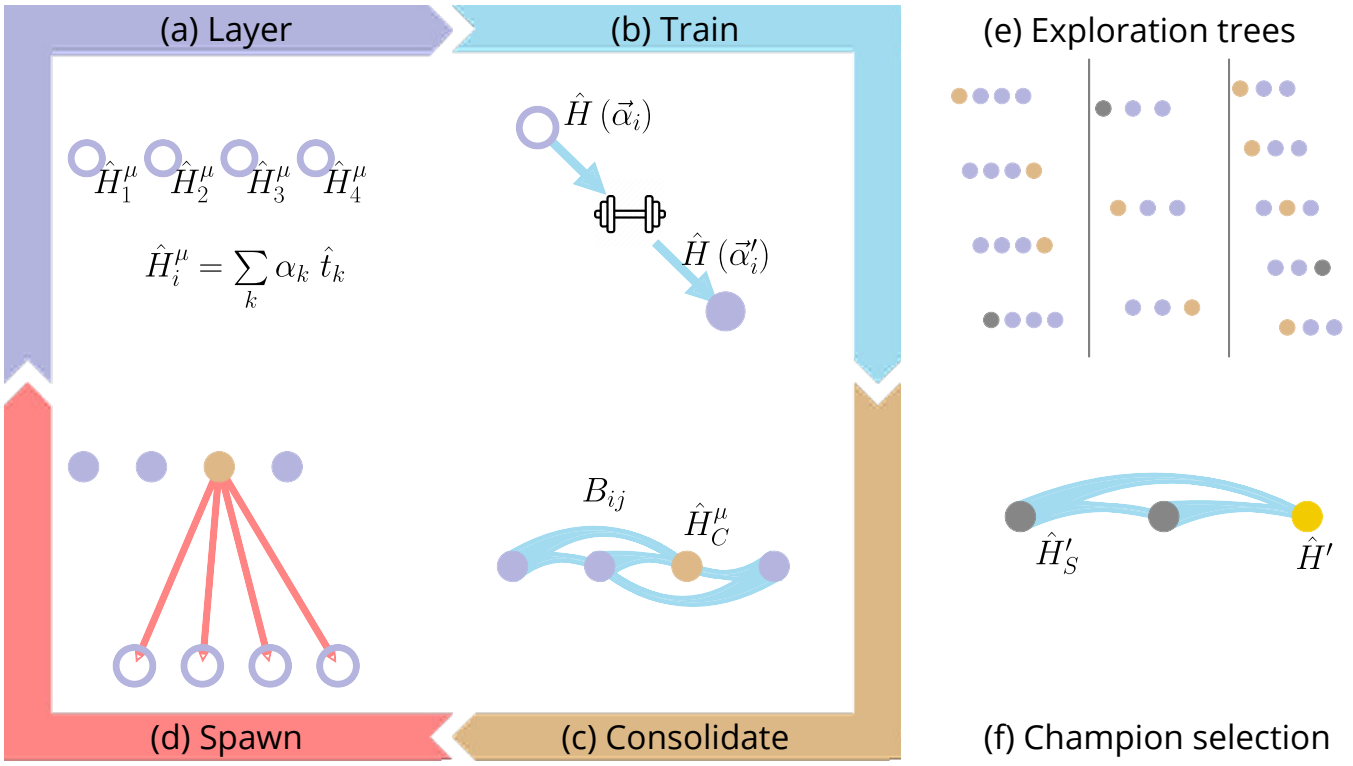

Figure 2.1: Schematic of Quantum Model Learning Agent (QMLA). **(a-d)** MS phase within an Exploration strategy (ES). **(a)** Models are placed as (empty, purple) nodes on the *active layer* μ, where each model is a sum of terms $\hat{t}_k$ multiplied by a scalar parameter α_k. **(b)** Each active model is trained according to a subroutine such as quantum Hamiltonian learning to optimise $\vec{\alpha}_i$, resulting in the trained (filled purple node) $\hat{H}(\vec{\alpha}'_i)$. **(c)** μ is consolidated, i.e. models are evaluated relative to other models on μ, according to the consolidation mechanism specificed by the ES. In this example, pairwise Bayes factors B_{ij} between $\hat{H}_i, \hat{H}_j$ are computed, resulting in the election of a single layer champion $\hat{H}_C^\mu$ (bronze). **(d)** A new set of models are *spawned* according to the chosen ES's model genertion strategy. In this example, models are spawned from a single parent. **(e-f)** Higher level of entire QMLA procedure. **(e)** The MS phase is presented on *exploration trees*. Multiple ES can operate in parallel, e.g. assuming different underlying physics. Each ES nominates a champion, $\hat{H}'_S$ (silver), after consolidating its branch champions (bronze). **(f)** $\hat{H}'_S$ from each of the above exploration trees are gathered on a single layer, which is consolidated to give the final champion model, $\hat{H}'$ (gold).

EXPLORATION STRATEGIES

QMLA is implemented by running N_t exploration trees (ETs) concurrently, where each ET corresponds to a unique MS and ultimately nominates a single model as its favoured approximation of $\hat{H}_0$. An ES is the set of rules which guide a single ET throughout its MS. We elucidate the responsibilities of ESs in the remainder of this section, but in short they can be summarised as:

i. model generation: combining the knowledge progressively acquired on the ET to construct new candidate models;

ii. decision criteria for the MS phase: instructions for how QMLA should respond at predefined junctions, e.g. whether to cease the MS after a branch has completed;

iii. true model specification: detailing the terms and parameters which constitute $\hat{H}_0$ (in the case where Q is simulated);

iv. modular functionality: subroutines called throughout QMLA are interchangeable such that each ES specifies the set of functions to achieve its goals.

QMLA acts in tandem with one or more ESs, through the process depicted in Fig. 2.2. In summary: QMLA sends a request to the ES for a set of models; ES designs models and places them as leaves on one of its branches, and returns the set $\mathbb{H}$; QMLA places $\mathbb{H}$ on a unique layer; QMLA trains the models in $\mathbb{H}$; QMLA consolidates $\mathbb{H}$; QMLA informs the ES of the results of training/consolidation of $\mathbb{H}$; ES decides whether to continue the search, and informs QMLA.

Model generation

The main role of any ES is to design candidate models to test against $\hat{H}_0$. This can be done through any means deemed appropriate, although in general it is sensible to exploit the information gleaned so far in the ET, such as the performance of previous candidates and their comparisons, so that successfel models are seen to *spawn* new models, e.g. by combining previously successful models, or by building upon them. Conversely, model generation can be completely determined in advance or entirely random. This alludes to the central design choice in composing an ES: how broad and deep should the searchable *model space* be, considering that adequately training each model is expensive, and that model comparisons are similarly expensive. The MS occurs within some , the size of which can usually be easily found by assuming that terms are binary – either the interaction they represent is present or not. If all possible terms are accounted for, and the total set of terms is $\mathcal{T}$, then there are $2^{|\mathcal{T}|}$ available candidates in the model space. The model space encompasses the closed[2]set of models construable by the set of terms considered by an ES. Because training models is slow in general, a central aim of QMLA is to search this space efficiently, i.e. to minimise the number of models considered, while retaining high quality models and providing a reasonable prospect of uncovering the true model, or a strong approximation thereof.

2 It is feasible to define an ES which uses an open model space, that is, there is no pre-defined $\mathcal{T}$, but rather the ES determines models through some other heuristic mechanism. In this thesis, we do not propose any such ES, but note that the QMLA framework facilitates the concept, see Chapter 3.

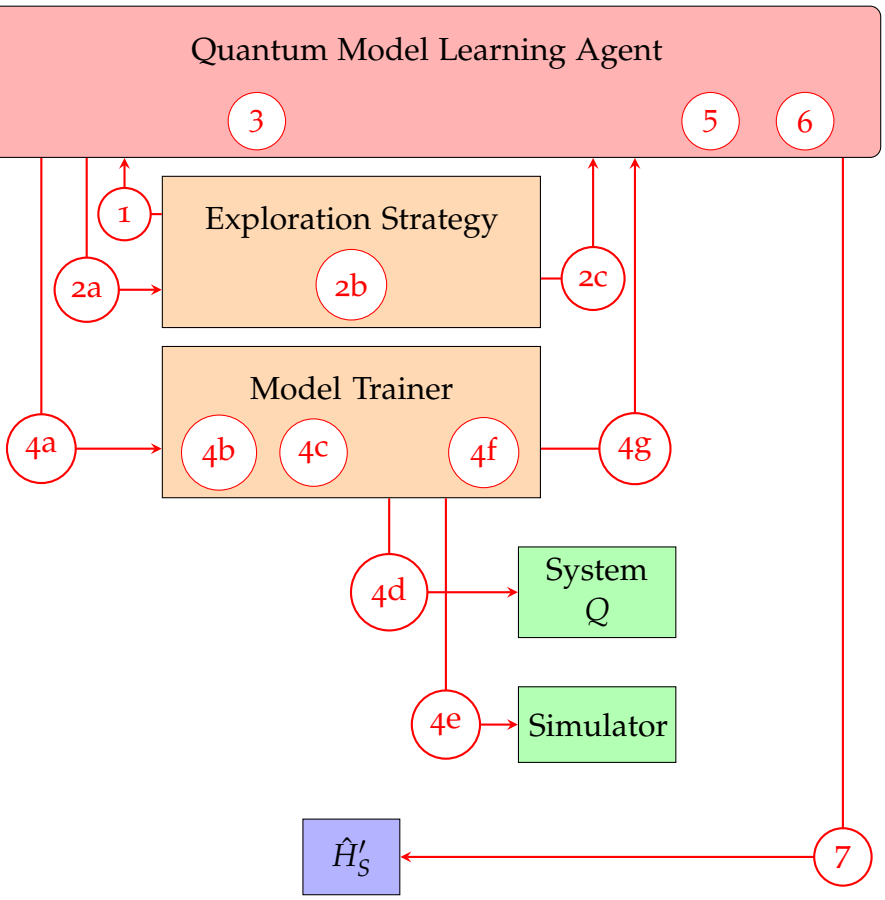

Figure 2.2: Interface between QMLA and a single ES. The main components are the ES, model training subroutine, target quantum system (i.e. black box, Q), and (quantum) simulator. The main steps of the algorithm, shown in red with arrows denoting data transferred during that step, are as follows. **1**, QMLA retrieves decision infrastructure from ES, such as the consolidation mechanism and termination criteria. **2**, models are designed/spawned; **2a**, QMLA signals to ES requesting a set of models, passing the results of the previous layers' models if appropriate. **2b**, ES spawns new models, $\mathbb{H}$; **2c**, ES passes $\mathbb{H}$ to QMLA. **3**, QMLA assigns a new layer $(\mu \leftarrow \mu + 1)$ and places the newly proposed models upon it. **4**, Model training subroutine (here quantum Hamiltonian learning), performed independently for each model $\hat{H}_i \in \mu$; **4a**, QMLA passes $\hat{H}_i$ to the model trainer; **4b**, construct a prior distribution P_i describing the model's parameterisation $\vec{\alpha}_i$; **4c**, design experiment e to perform on Q to optimise $\vec{\alpha}_i$; **4d**, perform e on Q to retrieve a datum d; **4e**, simulate e for particles $\{\vec{\alpha}_1, \ldots, \vec{\alpha}_N\}$ sampled from P_i to retrieve a *likelihood* l_e; **4e**, update the prior P_i based on (d, l_e). **5**, Evaluate and rank $\hat{H}_i \in \mu$ according to the ES's consolidation mechanism. **6**, Check ES's termination criteria; if reached, proceed to **(7)**, otherwise return to **(2)**. **7**, Nominate champion model, $\hat{H}'_S$.

Decision criteria for the model search phase

Further control parameters, which direct the growth of the ET, are set within the ES. At several junctions within Algs. 1, 2, QMLA queries the ES in order to decide what happens next. Here we list the important cases of this behaviour.

i. Parameter-learning settings
   (i) such as the prior distribution to assign each parameter during QHL, and the parameters needed to run SMC.
   (ii) the time scale on which to examine Q.
   (iii) the input probes to train upon.

ii. Branch comparison strategy
   (i) How to compare models within a branch (or QMLA layer). Some examples used in this work are (a) a points-ranking where each points are assigned according to Eqn. 2.8 (b) ranking reflecting each model's log-likelihood after training. All methods in §**??** can be thought of as branch comparison strategies.

iii. MS termination criteria
   (i) e.g. instruction to stop after a fixed number of iterations, or when a certain fitness has been reached.

iv. Champion nomination
   (i) when a single tree is explored, identify a single champion from the branch champions
   (ii) if multiple trees are explored, how to compare champions across trees.

True model specification

It is necessary also to specify details about the true model $\hat{H}_0$, at least in the case where QMLA acts on simulated data. Within the ES, we can set $\vec{T}_0$ as well as $\vec{\alpha}_0$. For example where the target system is an untrusted quantum simulator to be characterised, S_u, by interfacing with a trusted (quantum) simulator S_t, we decide some $\hat{H}_0$ in advance: the model training subroutine calls for likelihoods, those corresponding to $\hat{H}_0$ are computed S_u, while particles' likelihood are computed on S_t.

Modular functionality

Finally, there are a number of fundamenetal subroutines which are called upon throughout the QMLA algorithm. These are written independently such that each subroutine has a number of available implementations. These can be chosen to match the requirements of the user, and are set via the ES.

i. Model training procedure
   (a) i.e. whether to use QHL or quantum process tomography, etc.
   (b) In this work we always used QHL.

ii. Likelihood function: the method used to estimate the likelihood for use during quantum likelihood estimation within QHL, which ultimately depends on the measurement scheme.
   (a) By default, here we use projective measurement back onto the input probe state, $\left|\langle\psi| e^{-i\hat{H}t} |\psi\rangle\right|^2$.
   (b) It is possible to change this to implement any measurement procedure, for example an experimental procedure where the environment is traced out.

iii. Probes: defining the input probes to be used during training.
   (a) In general it is preferable to use numerous probes in order to avoid biasing particular terms.
   (b) In some cases we are restricted to a small number available input probes, e.g. to match experimental constraints.

iv. Experiment design heuristic: bespoke experiments to maximise the information on which models are individually trained.
   (a) In particular, in this work the experimental controls consist solely of $\{|\psi\rangle, t\}$.
   (b) Currently, probes are generated once according to iii., but in principle it is feasible to choose optimal probes based on available or hypothetical information. For example, probes can be chosen as a normalised sum of the candidate model's eigenvectors.
   (c) Choice of t has a large effect on how well the model can train. By default times are chosen proportional to the inverse of the current uncertainty in $\vec{\alpha}$ to maximise Fischer information, through the multi-particle guess heuristic [3]. Alternatively, times may be chosen from a fixed set to force QHL to reproduce the dynamics within those times' scale. For instance, if a small amount of experimental data is available offline, it is sensible to train all candidate models against the entire dataset.

v. Model training prior: change the prior distribution, e.g. Fig. 1.1(a)

Exploration strategy examples

To solidifiy the concept of ESs, and how they affect the overall, reach and runtime of a given ET, consider the following examples, where each strategy specifies how models are generated, as well as how trained models are compared within a branch. Recall that all of these strategies rely on QHL as the model training strategy, so that the run time for training, is $t_{\text{QHL}\sim N_e N_p t_U}$, where $t_U(n)$ is the time to compute the unitary evolution via the matrix exponential for an n-qubit model All models are trained using the default likelihood in Eq. (1.4). Assume the conditions

- all models considered are represented by 4-qubit models;
  - $t_{U(4)} \sim 10^{-3}$sec.
- each model undergoes a reasonable training regime;
  - $N_e = 1000, N_p = 3000$;
  - $\implies t_{\text{QHL}} = N_e \times N_p \times t_{U(4)} = 3000s \sim 1\text{h}$;
- Bayes factor calculations use
  - $N_e = 500, N_p = 3000$
  - $\implies t_{\text{BF}} \sim 2 \times 500 \times 3000 \times 10^{-3} \sim 1\text{h}$;
- there are 12 available terms
  - allowing any combination of terms, this admits a model space of size $2^{12} = 4096$
- access to 16 computer cores to parallelise calculations over
  - i.e. we can train 16 models or perform 16 BF comparisons in $1h$.

Then, consider the following model generation/comparison strategies.

a. Predefined set of 16 models, comparing every pair of models
   (i) Training takes $1h$, and $\binom{16}{2} = 120$ comparisons need $8h$
   (ii) total time is $9h$.

b. Generative procedure for model design, comparing every pair of models, running for 12 branches
   (i) One branch takes $9h \implies$ total time is $12 \times 9 = 108h$;
   (ii) total number of models considered is $16 \times 12 = 192$.

c. Generative procedure for model design, where less model comparisons are needed (say one third of all model pairs are compared), running for 12 branches
   (i) Training time is still $1h$
   (ii) One third of comparisons, i.e. 40 BF to compute, requires $3h$
   (iii) One branch takes $4h \implies$ total time is $36h$
   (iv) total number of models considered is also 192.

These examples illustrate some of the design decisions involved in ESs, namely whether timing considerations are more important than thoroughly exploring the model space. They also show considerable time–savings in cases where it is acceptable to forego all model comparisons. The approch in a. is clearly limited in its applicability, mainly in that there is a heavy requirement for prior knowledge, and it is only useful in cases where we either know $\hat{H}_0 \in \mathbb{H}$, or would be satisfied with approximating $\hat{H}_0$ as the closest available $\hat{H}_j \in \mathbb{H}$. On the opposite end of this spectrum, c. is an excellent approach with respect to minimising prior knowledge required by the algorithm, although at the significant expense of testing a much larger number of candidate models. There is no optimal strategy for all use–cases: specific quantum systems of

study demand particular considerations, and the amount of prior information available informs how wide the model search should reach.

In this work we have used two straightforward model generation routines. Firstly, during the study of various physical classes (Ising, Heisenberg, Fermi-Hubbard), a list of lattice configurations were chosen in advance, which were then mapped to Hamiltonian models. This ES is non-adaptive and indeed the model search consists merely of a single branch with no subsequent calls to the model generation routine, as in a.. In the latter section instead we use a genetic algorithm (GA): this is clearly a far more general strategy, at a significant computational cost, but is suitable for systems where we have less knowledge in advance. In this case, new models are designed based heavily on results from earlier branches of the ET. The genetic algorithm model generation subroutine is listed in Alg. 3, and can broadly be summed up thus: the best models in a generation μ produce offspring which constitute models on the next generation. These types of evolutionary algorithms ensure that newly proposed candidates inherit some of the structure which rendered previous candidates (relatively) successful, in the expectation that this will yield ever-stronger candidates.

GENERALITY

Several aspects of QMLA are deliberately vague in order to facilitate generality.

- *Model* can mean any description of a quantum system which captures the interactions it is subject to.
  - Here we exclusively consider Hamiltonian models, but Lindbladian models can also be considered as generators of quantum dynamics.
- *Model training* is any subroutine which can train a given model, i.e. optimise a given parameterisation under the assumption that represents the target system.
  - Currently only QHL has been used, but tomography is valid in principle, albeit slower.
  - QHL relies on the calculation of a characteristic likelihood function; this too is not restricted to the generic form of Eq. (1.4) and can be replaced by any form which represents the likelihood that experimental conditions e result in measurement datum d. We will see examples of this in **??** where we trace out part of the system in order to represent open systems.
- *Model selection*, or *consoldation* can be as rigourous as desired by the user.
  - Consolidation occurs at the branch level of each ET, but also in finding the tree champion, and ultimately the global champion.
  - In practice, we use either BF or a related concept such as TLTL which are statistically signicative. However, in **??** we will consider a number of alternative schemes for discerning the strongest models.

Agency

While the concept of *agency* is contentious [21], we can view our overall protocol as a multi-agent system [22], or even an agent based evolutionary algorithm [23], because any given ES satisfies the definition, *the population of individuals can be considered as a population of agents*, where we mean the population of models present on a given ET. More precisely, we can view individual models as *learning agents* according to the criteria of [24], i.e. that a learning agent has

- a *problem generator*: designs actions in an attempt to learn about the system – this is precisely the role of the EDH;
- a *performance element*: implements the the designed actions and measures the outcome – the measurement of a datum following the experiment chosen by the EDH;
- a *critic*: the likleihood function informs whether the designed action (experiment) was successful;
- a *learning element*: the updates to the weights and overall parameter distribution improve the model's performance over time.

We depict this analogy in Fig. 2.3. Finally, the model design strategy encoded in the ES *can* allow agency, by permitting the spawn rules autonomy, so we label the entire procedure as the quantum model learning agent.

ALGORITHMS

We conclude this chapter by listing the algorithms used most frequently, in order to clarify each of their roles, and how they interact. Alg 1 shows the overall QMLA algorithm, which is simplified greatly to a loop over the MS of each ES. The MS iteself is listed in Alg. 2, which contains calls to subroutines for model learning (QHL, Alg. 4), branch evaluation (which can be based upon BF, Alg. 5) and centers on the generation of new models, an example of which – based on a genetic algorithm – is given in Alg. 3.

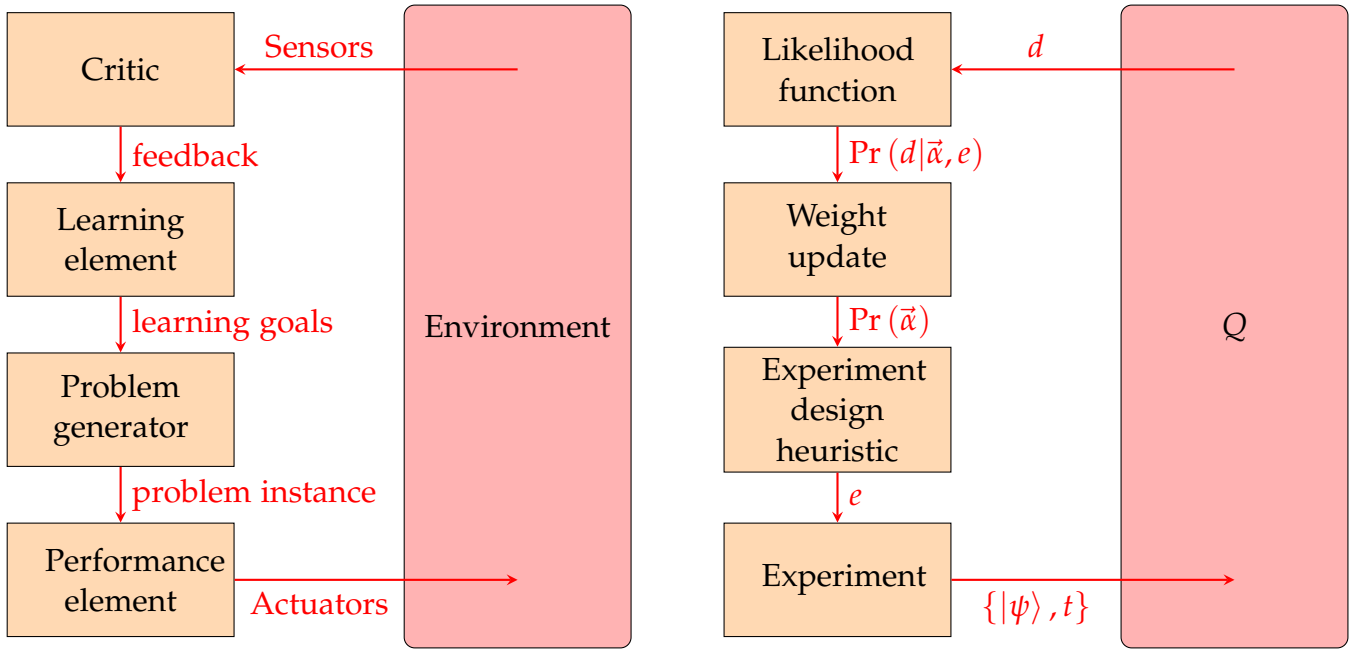

Figure 2.3: Learning agents. **Left**: definition of a learning agent, where an *environment* is affected by *actuators* which realise a *problem instance*, designed by a *problem generator*, through some *performance element*. The result of the agent's action is detected by *sensors*, which the *critic* interprets with respect to the agent's *learning goals*, by providing *feedback* to the *learning element*. *Right*: mapping of the concept of a learning agent on to an individual model. A target quantum system, Q, is queried by performing some experiment e, designed by an experiment design heuristic, and implemented by evolving a probe state $|\psi\rangle$ for time t. The systems is measured, and the datum d is sent to the likelihood function, which sends the likelihood $\Pr(d|\vec{\alpha}, t)$ to the weight update (and the parameter distribution update), before designing another experiment.

Algorithm 1: Quantum Model Learning Agent

Input: Q `// some physically measurable or simulateable quantum system`
Input: $\mathbb{S}$ `// set of exploration strategies`

Output: $\hat{H}'$ `// champion model`

$\mathbb{H}_c \leftarrow \{\}$
for $S \in \mathbb{S}$ **do**
  $\hat{H}'_S \leftarrow$ `model_search(Q, S)`
  $\mathbb{H}_c \leftarrow \mathbb{H}_c \cup \{\hat{H}'_S\}$ `// add ES champion to collection`
end
$\hat{H}' \leftarrow$ `final_champion(`$\mathbb{H}_c$`)`
return $\hat{H}'$

Algorithm 2: ES subroutine: `model_search`

Input: Q `// some physically measurable or simulateable quantum system`
Input: S `// Exploration strategy: collection of rules/subroutines`

Output: $\hat{H}'_S$ `// Exploration strategy's nominated champion model`

$\nu \leftarrow \{\}$
$\mathbb{H}_c \leftarrow \{\}$
while *!S.terminate()* **do**
  $\mu \leftarrow$ `S.generate_models(`ν`)` `// e.g. Alg. 3`
  for $\hat{H}_i \in \mu$ **do**
    $\hat{H}'_i \leftarrow$ `S.train(`$\hat{H}_i$`)` `// e.g. Alg. 4`
  end
  $\nu \leftarrow$ `S.evaluate(`μ`)` `// e.g. pairwise via Alg. 5`
  $\hat{H}^{\mu}_c \leftarrow$ `S.branch_champion(`ν`)` `// use` ν `to select a branch champion`
  $\mathbb{H}_c \leftarrow \mathbb{H}_c \cup \{\hat{H}^{\mu}_c\}$ `// add branch champion to collection`
end
$\hat{H}'_S \leftarrow$ `S.nominate_champion(`$\mathbb{H}_c$`)`
return $\hat{H}'_S$

Algorithm 3: ES subroutine: `generate_models` (example: genetic algorithm)

Input: ν // information about models considered to date

Input: $g(\hat{H}_i)$ // objective function

Output: $\mathbb{H}$ // set of models

$N_m = |\nu|$ // number of models
for $\hat{H}_i \in \nu$ **do**
  $g_i \leftarrow g(\hat{H}_i)$ // model fitness via objective function
end
$r \leftarrow \texttt{rank}(\{g_i\})$ // rank models by their fitness
$\mathbb{H}_t \leftarrow \texttt{truncate}(r, \frac{N_m}{2})$ // truncate models by rank: only keep $\frac{N_m}{2}$
$s \leftarrow \texttt{normalise}(\{g_i\})\ \forall \hat{H}_i \in \mathbb{H}_t$ // normalise remaining models' fitness
$\mathbb{H} = \{\}$ // new batch of chromosomes/models
while $|\mathbb{H}| < N_m$ **do**
  $p_1, p_2 = \texttt{roulette}(s)$ // use s to select two parents via roulette selection
  $c_1, c_2 = \texttt{crossover}(p_1, p_2)$ // produce offspring models
  $c_1, c_2 = \texttt{mutate}(c_1, c_2)$ // probabilistically mutate
  $\mathbb{H} \leftarrow \mathbb{H} \cup \{c_1, c_2\}$ // add new models to batch
end
return $\mathbb{H}$

Algorithm 4: Quantum Hamiltonian Learning

Input: Q // some physically measurable or simulatable quantum system, described by $\hat{H}_0$
Input: $\hat{H}_i$ // Hamiltonian model attempting to reproduce data from $\hat{H}_0$
Input: $\Pr(\vec{\alpha})$ // probability distribution for $\vec{\alpha} = \vec{\alpha}_0$
Input: N_e // number of epochs to iterate learning procedure for
Input: N_p // number of particles to draw from $\Pr(\vec{\alpha})$
Input: $\Lambda(\Pr(\vec{\alpha}))$ // Heuristic algorithm which designs experiments
Input: $\mathtt{RS}(\Pr(\vec{\alpha}))$ // Resampling algorithm for redrawing particles
Output: $\vec{\alpha}'$ // estimate of Hamiltonian parameters

Sample N_p times from $\Pr(\vec{\alpha}) \leftarrow \mathcal{P}$ // particles

for $e \in \{1 \rightarrow N_e\}$ **do**
  $t, |\psi\rangle \leftarrow \Lambda(\Pr(\vec{\alpha}))$ // design an experiment
  for $p \in \mathcal{P}$ **do**
    Retrieve particle $p \leftarrow \vec{\alpha}_p$
    Prepare Q in $|\psi\rangle$, evolve and measure after $t \leftarrow d$ // datum
    $|\langle d| e^{-iH(\vec{\alpha}_p)t} |\psi\rangle|^2 \leftarrow Pr(d|\vec{\alpha}_p; t)$ // likelihood
    $w_p \leftarrow w_p \times Pr(d|\vec{\alpha}_p; t)$ // weight update
  end
  if $1/\sum_p w_p^2 < N_p/2$ // check whether to resample (are weights too small?)
  then
    $\mathtt{RS}(\Pr(\vec{\alpha})) \leftarrow \mathcal{P}$ // Redraw particles via resampling algorithm
  end
end
$\mathtt{mean}(\Pr(\vec{\alpha})) \leftarrow \vec{\alpha}'$
return $\vec{\alpha}'$

Algorithm 5: Bayes Factor calculation

Input: Q // some physically measurable or simulateable quantum system.
Input: $\hat{H}'_j, \hat{H}'_k$ // Hamiltonian models after QHL (i.e. $\vec{\alpha}_j, \vec{\alpha}_k$ already optimised), on which to compare performance.
Input: $\mathcal{E}_j, \mathcal{E}_k$ // experiments on which $\hat{H}'_j$ and $\hat{H}'_k$ were trained during QHL.
Output: B_{jk} // Bayes factor between two candidate Hamiltonians
$\mathcal{E} = \{\mathcal{E}_j \cup \mathcal{E}_k\}$
for $\hat{H}'_i \in \{\hat{H}'_j, \hat{H}'_k\}$ **do**
  $\mathcal{L}_i = 0$ // total log-likelihood of $\hat{H}_i$
  for $e \in \mathcal{E}$ **do**
    $e \leftarrow t, |\psi\rangle$ // assign time and probe from experiment control set
    Prepare Q in $|\psi\rangle$, evolve and measure after $t \leftarrow d$ // datum
    $\left|\langle d| e^{-i\hat{H}'_i t} |\psi\rangle\right|^2 \leftarrow Pr(d|\hat{H}_i, t)$ // total likelihood for $\hat{H}'_i$ on e
    $log\left(Pr(d|\hat{H}_i, t)\right) \leftarrow l_e$ // log total likelihood for $\hat{H}'_i$ on e
    $\mathcal{L}_i + l_e \leftarrow \mathcal{L}_i$ // add l_e to total log total likelihood
  end
end
$\exp\left(\mathcal{L}_j - L_k\right) \leftarrow B_{jk}$ // Bayes factor between models

return B_{jk}

3

SOFTWARE

All of the details in Chapter 1 and Chapter 2 are implemented in the QMLA software framework, a (mostly) Python codebase which underlies all of the arguments, results and figures in this thesis. The codebase is designed to simplify the process of running QMLA or QHL on novel systems. In particular, the core QMLA algorithm can support a wide range of ESs, allowing for the design of bespoke ESs to account for the specific requirements of any given system. In this chapter we give an overview of the QMLA software, implementation and instructions for its use.

IMPLEMENTATION

In this section we describe the technical details of the implementation of the algorithm described in Chapter 2, as well as a number of relevant subroutines. We do not introduce new concepts, so readers interested in applications of the techniques may prefer to skip to **??**

Object oriented programming

We first introduce the concepts of object-oriented programming, and in particular *inheritance* between objects, since this will feature in later discussion about the implementation of QMLA and ESs. Python is a robust object-oriented language [25], meaning that we can frame concepts as objects which permit actions to be performed by/to them. In particular, objects in Python are formulated as *classes*, which can have associated *attributes* and *methods*. For example, we can encode the concept of a footballer as an object, such that the player object has attributes, e.g. number of games played and goals scored in a season, as well as methods for specific calculations, We can then utilise the footballer class to store information about an individual player, such as their team and goals scored. A fundamental concept in object-oriented programming is *inheritance* between objects, such that a *child* objects inherit properties of its *parent*. In general, a parent object can be thought of as an abstract concept, which provides basic functionality and reasonable default properties, while a child object can specify further details. For example, an Athlete class can act as a parent to the footballer class, where the Athlete class holds core information such as date of birth. This allows for the Athlete class to be recycled as the *base* class for other child classes which have the same underlying requirements, e.g RubgyPlayer. We list this example in Listings 3.1 to 3.2.

```
class Athlete():
```

```
    def __init__(
        self,
        name,
        birth_day,
        birth_month,
        birth_year,
    ):
        # Use information given
        self.name = name
        self.date_of_birth = datetime.date(
            birth_year, birth_month, birth_day
        )

    def age(self, round_down=True):
        days_since_birth = datetime.date.today() - self.
           date_of_birth
        age = days_since_birth.days / 365

        if round_down:
            age = int(age)

        return age

    def summary(self):
        summary = "{name} is a {age}-year old athlete.".format(
            name = self.name,
            age = self.age()
        )
        print(summary)


bob = Athlete(
    name='Bob',
    birth_day = 11,
    birth_month = 11,
    birth_year = 1993,
)
bob.summary()
```

Listing 3.1: "Parent class

```
class Footballer(Athlete):
    def __init__(
        self,
        footed,
        team,
        size = 'medium',
        **kwargs
    ):
        # Pass arguments to the parent class
        super().__init__(**kwargs)

        # Use information given
        self.team = team
        self.footed = footed
        self.size = size

        # Default attributes
        self.goals_scored = 0

    def summarise(self):
        summary = "{size} {player} plays for {team} and has
           scored {num_goals} goals.".format(
            size = self.size,
            player = self.name,
            team = self.team,
            num_goals = self.goals_scored
        )
        print(summary)

    def record_goals(self, num_new_goals):
        self.goals_scored += num_new_goals


mickey = Footballer(
    name = 'Mickey',
    footed = 'left',
    team = 'QECDT–FC',
    birth_day = 1,
    birth_month = 1,
```

```
    birth_year = 1990,
    size = 'Big'
)
mickey.record_goals(num_new_goals = 10)
mickey.summarise()
```

Listing 3.2: Child class

PYTHON FRAMEWORK

A driving motivation for the development of QMLA is generality: we endevaour to make QMLA applicable to any target quantum system. We provide a framework, where users can tailor the inputs and methodology to their needs: we depict the main components of the framework in Fig. 3.1, broadly grouping concepts as part of its *infrastructure*, *algorithm* or *application*. In short, users need only specify the elements of the framework in the *application* segment, without concern for the underlying mechanics of QMLA; in particular, users interface with the framework through the design of a bespoke ES, described next.

Application

The application of QMLA refers to the choice of target system, and how QMLA searches the MS in attempt to discover its model. As outlined in Section 2.4, ESs play the role of defining QMLA's objectives, guiding the steps it takes, and designing the models to be tested. We facilitate the study of any system by providing a robust `ExplorationStrategy` base class, with all of the functionality expected of a generic ES, allowing users to inherit and build upon it. In particular, ESs allow users to specify the implementation of aspects listed in Section 2.4, as well as further details.

Modular functionality

The most crucial methods[1] of the ES class are modular, meaning that they can be directly replaced, provided the alternative method fulfils the same role. Our base ES class uses sensbile defaults for this modular functionality, but this flexible mechanism allows for adapting QMLA by choosing an approach for each of the following subroutines.

- Likelihood function. By default, QHL calls a subroutine to compute Eq. (1.4). This can be replaced by any function which, given a Hamiltonian, evolution time and probe state, returns the likelihood, according to the experiment you wish to simulate. For example, in **??**, the data on which models are trained comes from experimental measurements, so

1 Methods are functions which are associated with classes.

we replace the likelihood function with a calculation corresponding to the experimental procedure.

- Probe generation. The training phase requires a set of probes against which to optimise individual models. Users may wish to specify the design of such probes, for example to match experimental constraints which restrict the realisible probes in the performance of the experiment.
- Experiment design heuristic (EDH). The choice of EDH greatly influences how the training will perform. We provide a base class implementing PGH, as well as child classes for each of the EDHs listed in Section 1.5.2.
- Prior. The method of drawing the prior distribution can be replaced, for example, with a method for constructing a uniform distribution on each parameter.

Additionally, applications require a series of settings for the model training phase, such as the hyperparameters required by the resampling algorithm, [6], as well as detailing the true (target) model, $\hat{H}_0$, in the case where Q is a simulated quantum system. We can also specify some ES-specific analyses to examine its internal performance, although this is generally required during development/testing, and less useful thereafter.

Algorithm

The algorithm layer of Fig. 3.1 implements the core steps of QMLA, as shown in Fig. 2.1, by running a set of exploration trees (ETs), each of which communicate with a unique ES. The core QMLA class manages the database of models and their comparisons, and decides how to react at certain stages, by consulting the decision criteria set by the ES.

Parallel implementiation

The implementation of QMLA seeks to separate the organisation of the MS from the cumbersome calculations which enable the search. We can offload those calculations to a compute cluster (server) to run *in parallel*, allowing for significant speedup of the entire QMLA procedure. QMLA distributes jobs to worker processes in a server, i.e. we assume that QMLA is run on a machine with N_c available parallel *processes*[2]. Then, the expensive calculations, namely training and comparing models, are not performed directly within the QMLA class, but instead are farmed out across the server. The role of QMLA then is to collate the outcome of those calculations in conjunction with the set of ETs, until each ET is deemed complete, and then to consolidate the set of ET champions, ultimately setting the global champion, $\hat{H}'$. Thereafter it can perform some analysis, e.g. to generate a series of plots which demonstrate how the model search progressed, as well as the evidence in favour of $\hat{H}'$, including for example the reproduction of Q's dynamics by $\hat{H}'$. See **??** for further details.

2 Note when running in *serial* (e.g. running locally on a personal machine), it is valid to simply set $N_c = 1$.

While there are a number of strategies for parallelising code over a cluster, we use the *master-worker* strategy, where one process acts as the *master*, determining which calculations are required at any given moment, then brokering self-contained tasks to *workers*, which blindly solve a small problem, without knowledge of the wider context or algorithm [26]. The mapping here is trivial: the master of our algorithm is QMLA, while workers can be used for the training and comparison of models. The QMLA class is hence assigned a single process solely for its considerations, e.g. for the ranking of models and determination of the next models to tests, while the remaining $N_c - 1$ processes lay dormant until QMLA requests that they perform a job.

We use a simple *task queue* for the distribution of jobs, e.g. after it has decided to train models $\hat{H}_i, \hat{H}_j, \hat{H}_k$, QMLA defines a job per model, where each job refers to the training of a single model, and places those jobs on a shared job queue, whereupon any available worker can take the next job from the queue and compute it. There are two types of task for workers:

- to train a candidate model $\hat{H}_i$: the worker first requests some essenital information about the model from the databse, e.g. the name, terms and prior associated with the model, packaged in M_i; following completion, the worker compresses the result, R_i, and sends it to the database for storage.
- to compare two models, $\hat{H}_i, \hat{H}_j$: the worker retrieves R_i, R_j from the database, performs the calculation, and returns the compressed outcome of the comparison, C_{ij}, to the database.

The master QMLA class can also access said database, and copies the compressed results packets R_i and C_{ij}, in order to account for the results in its decision-making. It is worth noting that tasks are completely independent, so some worker processes may compute comparisons while others train models simultaneously, although obviously the comparison can not begin until both R_i, R_j are available. This is dealt with easily by using a *blocking* protocol, where new batches of jobs are not released until the master receives all the results of jobs on which the new tasks depend. QMLA simply waits until all models on a given layer have been trained before queueing comparisons on that layer, to ensure a comparison can not start without the data needed to compute that comparison. We use a redis database and task queue [27, 28]. We depict the structure of our parallel architecture, and the master-worker strategy, in Fig. 3.2.

Infrastructure

The infrastructure enabling the distribution of QMLA's tasks across a set of worker processes can be summarised as a set of classes representing the objects on which we must perform expensive calculations, as well as functions to launch those calculations independently of any other calculation, and a database which can be accessed by QMLA and all workers.

We need a series of distinct classes to represent models, for use in each stage of QMLA: a *trainable* class is used for the parameter optimisation, while *comparable* classes are used for computing BFs, Crucially, this separation allows us to perform data-heavy calculations inde-

pendently, e.g. on a remote process within a compute cluster, and discard the class instance used for the calculation, while only the relatively small *storage* classs is retained by QMLA for later use. To achieve this separation between calculation and analysis, we use a redis database, which holds compressed summaries of the outcomes of tasks.

Analyses and plots

USAGE

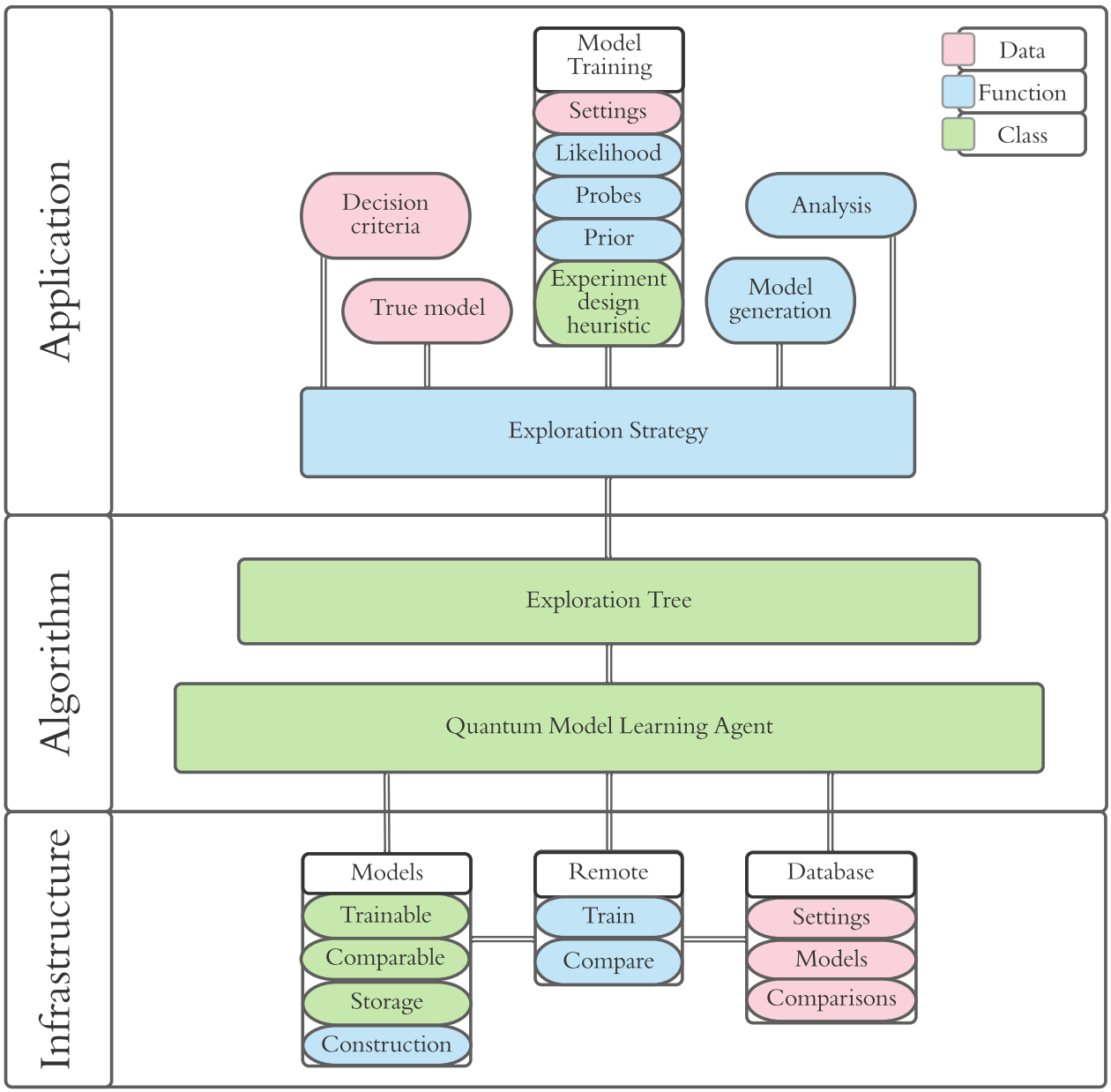

Figure 3.1: Overview of important *objects* in the QMLA codebase. Colours encdoe the type of object: red objects are *data*, blue are *functions/methods* and green are *classes*. Objects are grouped broady, with double lines showing communication channels between (groups of) objects. *Infrastructure*: functions for the implementation of model training/comparisons on a remote server. *Algorithm*: implementation of the iterative procedures and decision-making laid out in Chapter 2. *Application*: inter-changeable data/functionality for the unique requirements of a given target system. Users wishing to customise QMLA must choose a valid object for each of those in *applications*, and need not alter any of the underlying framework.

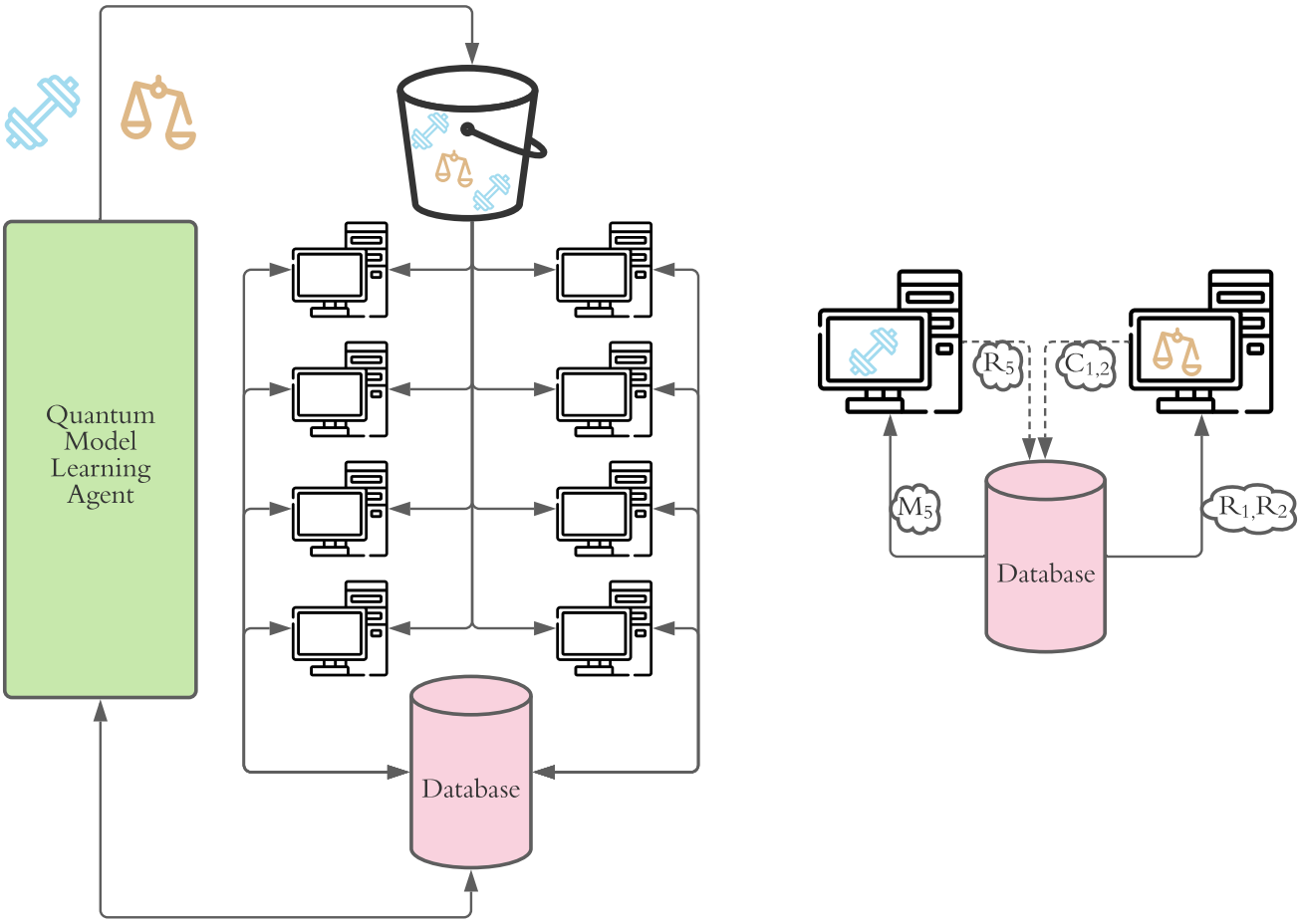

Figure 3.2: Parallel architecture for QMLA. **Left**, QMLA generates tasks – either to train (blue dumbbells) or compare (oragne scales) models – and places them in a task queue. Worker processes (depicted as computers) retrieve those tasks and compute them in parallel, and interact with a database. **Right**, Distributed tasks occuring in parallel. The left-hand process assumes the task of training the model with ID 5: it first queries the database for a packet of core information, M_5, which informs the model training procedure, for example the terms and parameters of model 5. After training, it sends a packet, R_5, summarising the result of model 5's training. The right-hand process compares two models with IDs 1 and 2, by first retrieivng the results packets R_1, R_2, then storing the comparison $C_{1,2}$ on the database.

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] 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.

[5] 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.

[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] Rodolfo A Jalabert and Horacio M Pastawski. Environment-independent decoherence rate in classically chaotic systems. *Physical review letters*, 86(12):2490, 2001.

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

[19] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[20] Sheng-Tao Wang, Dong-Ling Deng, and Lu-Ming Duan. Hamiltonian tomography for quantum many-body systems with arbitrary couplings. *New Journal of Physics*, 17(9):093017, 2015.

[21] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer, 1996.

[22] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[23] Ruhul A Sarker and Tapabrata Ray. Agent based evolutionary approach: An introduction. In *Agent-Based Evolutionary Search*, pages 1–11. Springer, 2010.

[24] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[25] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[26] Roger W Hockney and Chris R Jesshope. *Parallel Computers 2: architecture, programming and algorithms*. CRC Press, 2019.

[27] Redis, Nov 2020. [Online; accessed 7. Nov. 2020].

[28] RQ: Simple job queues for Python, Nov 2020. [Online; accessed 7. Nov. 2020].