Quantum Model Learning Agent Documentation

Release 1

Brian Flynn, Antonio Andreas Gentile, Raffaele Santagati

Oct 27, 2021

CONTENTS

CHAPTER

ONE

GLOSSARY

Bayes factor Statistical measure of performance between two models at explaining the same dataset

BF

See also:

Bayes factor

EDH

ES

See also:

Exploration Strategy

ET

See also:

Exploration Tree

Experiment Design Heuristic Mechanism to design informative experiments to perform upon the *system* from which the model training can learn. Defined in *Experiment design heuristic*.

Exploration Strategy The mechanism by which a tree grows, specifying new models to consider, when to stop considering new models, how to remove models, etc.

See also:

Defined in *Exploration Strategy*.

Exploration Tree Unique tree associated with an individual *Exploration Strategy*.

See also:

Defined in *Structure*.

global champion Single model favoured by *Quantum Model Learning Agent* as the strongest candidate to represent the *system*.

Instance A single implementation of QMLA.

See also:

Defined in *Structure*.

Probe Input state evolved by both the candidate model and system to draw comparisons between them. Defined in *Probes*.

QHL

See also:

Quantum Hamiltonian Learning

QLE

See also:

Quantum Likelihood Estimation

QMLA

See also:

Quantum Model Learning Agent

Quantum Hamiltonian Learning Algorithm for learning the parameters of a given model.

Quantum Likelihood Estimation Algorithm used to perform Bayesian inference during *QHL*

Quantum Model Learning Agent Algorithm/framework for finding model of quantum system.

Run A collection of *instance* s. Note that for a run, all instances must target the same *system* .

See also:

Defined in *Structure*.

Run Results Directory Directory to which results are stored for an individual *run*. Consists of results files for all the *instance* s in the run, as well as analyses on the model, *instance* and *run* levels.

System The target system, i.e. underlying model. In simulation, this is used to generate the expectation values against which likelihood estimation occurs. In experiments, the form of the system is unknown, but data obtained from experiments are used in the likelihood estimation instead.

True Model

See also:

system

CHAPTER

TWO

OVERVIEW

Quantum Model Learning Agent (QMLA) is a machine learning protocol for the characterisation of quantum mechanical systems and devices. It aims to determine the model which best explains observed data from the system under study. It does this by considering a series of candidate models, performing a learning procedure to optimise the performance of those candidates, and then selecting the best candidate. New candidate models can be constructed iteratively, using information gained so far in the procedure, to improve the model approximation.

QMLA can operate on simulated or experimental data, or be incorporated with an online experiment. This document provides details on QMLA's mechanics, and provides examples of usage. Particular attention is paid to the concept and design of *Exploration Strategies (ES)*, the primary mechanism by which users ought to interact with the software. Custom *ES* can be developed and plugged into the *QMLA* framework, in order to target systems, run experiments and generate models according to the user's requirements. Several aspects of the framework are modular, allowing users to select the combination which suits their requirements, or easily add functionality as needed.

This chapter briefly introduces the core concepts at a high level; thorough detail can be found in *User Guide*.

2.1 Models

Models encapsulate the physics of a system. We generically refer to *models* because *QMLA* is indifferent to the formalism employed to describe the system. Usually we mean *Hamiltonian* models, although *QMLA* may also be used to learn *Lindbladian* models.

Models are simply the mathematical objects which can be used to predict the behaviour of a system, uniquely represented by a parameterisation. Each term in a model is really a matrix corresponding to some physical interaction; each such term is assigned a scalar parmeter. The total model is a matrix, which is computed by the sum of the terms multiplied by their parameters. For example, 1-qubit models can be constructed using the Pauli operators $\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z$, e.g. $\hat{H}_{xy} = \alpha_x \hat{\sigma}_x + \alpha_y \hat{\sigma}_y$. Then, $\hat{H}_{xy}$ is completely described by the vector $\boldsymbol{\alpha} = (\alpha_x, \alpha_y)$, when we know the corresponding terms $\boldsymbol{T} = (\hat{\sigma}_x, \hat{\sigma_y})$. In general then, models are given by $\hat{H}(\boldsymbol{\alpha}) = \boldsymbol{\alpha} \cdot \boldsymbol{T}$.

In the Hamiltonian (closed) formalism, terms included in the model correspond to interactions between particles in the system. For example, the Ising model Hamiltonian on N sites (spins), $\hat{H}^{\otimes N} = J \sum_{i=1}^{N-1} \hat{\sigma}_i^z \hat{\sigma}_{i+1}^z$, includes terms $\hat{\sigma}_i^z \hat{\sigma}_{i+1}^z$ which are the interactions between nearest neighbour sites $(i, i+1)$.

QMLA reduces assumptions about which interactions are present, for instance by considering models $\hat{H}^{\otimes 5}$ and $\hat{H}^{\otimes 8}$, and determining which model (5 or 8 spins) best describes the observed data. Moreover, *QMLA* facilitates consideration of all terms independently, e.g. whether the system is better described by a partially connected Ising lattice $\hat{H}_1$ or a nearest-neighbour connected Ising chain $\hat{H}_2$:

$$\hat{H}_1 = \alpha_1 \hat{\sigma}_1^z \hat{\sigma}_2^z + \alpha_2 \hat{\sigma}_1^z \hat{\sigma}_3^z + \alpha_3 \hat{\sigma}_1^z \hat{\sigma}_4^z$$

$$\hat{H}_2 = \beta_1 \hat{\sigma}_1^z \hat{\sigma}_2^z + \beta_2 \hat{\sigma}_2^z \hat{\sigma}_3^z + \beta_3 \hat{\sigma}_3^z \hat{\sigma}_4^z$$

Then, models exist in a *model space*, i.e. the space of all valid combinations of the available terms. Any combination of terms is permissible in a given model. *QMLA* can then be thought of as a search through the model space for the set of terms which produce data that best matches that of the system. Since these terms correspond to the physical interactions affecting the system, the outcome can be thought of as a complete characterisation.

2.2 Model Training

Model traning is the process of optimising the parameters α of a given model against the system's data. The model which is being learned does not need to be the *true* model; any model can attempt to describe any data. A core hypothesis of *QMLA* is that models which better reflect the true model will produce data more consistent with the system data, when compared against less-physically-similar models.

In principle, any parameter-learning algorithm can fulfil the role of training models in the *QMLA* framework, but in practice, *Quantum Hamiltonian Learning (QHL)* is used to perform Bayesian inference on the parameterisation, and hence attempt to find the optimal parameterisation for each model [WGFC13a], [WGFC13b], [GFWC12]. This is performed using [QInfer].

2.3 Model Comparison

Two candidate models $\hat{H}_1, \hat{H}_2$, having undergone model training, can be compared against each other to determine which one better describes the system data. *Bayes factor (BF)* provide a quantitative measure of the relative strength of the models at producing the data. We take the *BF* $B(\hat{H}_1, \hat{H}_2)$ between two candidate models as evidence that one model is preferable. Evidence is compiled in a series of pairwise comparisons; models are compared with a number of competitors such that the strongest model from a pool can be determined as that which won the highest number of pairwise comparisons.

2.4 Structure

QMLA is structured over several levels:

Models are individual candidates (e.g. Hamiltonians) which attempt to capture the physics of the *system*.

Layers/Branches: models are grouped in layers, which are thought of as branches on exploration trees.

Exploration trees are the objects on which the model search takes place: we think of models as *leaves* on *branches* of a tree. The model search is then the effort to find the single leaf on the tree which best describes the *system*. They grow and are pruned according to rules set out in the exploration strategy.

Exploration Strategies (*ES*) are bespoke sets of rules which decide how *QMLA* ought to proceed at each step. For example, given the result of training/comparing a previous set of models, the *ES* determines the next set of candidate models to be considered.

Instance: a single implementation of the *QMLA* protocol, whether to run the entire model search or another subroutine the framework. Within an instance, several exploration trees can grow independently in parallel: we can then think of *QMLA* as a search for the single best leaf among a forest of trees, each of which corresponds to a unique exploration strategy.

Run many instances which pertain to the same problem. *QMLA* is run independently for a number of instances, allowing for analysis of the algorithm's performance overall, e.g. that *QMLA* finds a particular model in 50% of 100 instances.

2.5 Outputs

QMLA automatically performs a series of analyses and produces associated plots. These are stored in a unique folder generated for the *run* upon launch: this folder is specified by the date and time of the launch and is located, relative to the *QMLA* main project directory in, e.g., `launch/results/Jan_01/12_34`. These are detailed in *Output and Analysis*.

2.6 User Interface

In order to tailor *QMLA* to a user's needs, they must design a bespoke *Exploration Strategy*. That is, the user must write a class building upon and inheriting from `ExplorationStrategy`, encompassing all of the logic required to achieve their use case, for example by incorporating a genetic algorithm within the method called upon for constructing new candidates, `generate_models()`. Then, that class must be available to `get_exploration_class()`, by ensuring it is included in one of the `import` statements in `qmla/exploration_strategies/__init__.py`. Finally, instruct *QMLA* to use that *ES* for a run in the launch script (see *Launch*). These steps are laid out in full in section_tutorial.

CHAPTER

THREE

USER GUIDE

3.1 Quantum Model Learning Agent

The class which controls everything is `QuantumModelLearningAgent`. An *instance* of this class is used to run one of the available algorithms; many independent instances can operate simultaneously and be analysed together (e.g. to see the *average* reproduction of dynamics following model learning). This is referred to as a *run* The *QMLA* class provides methods for each of the available algorithms, as well as routines required therein, and methods for analysis and plotting. In short, the available algorithms are

`Quantum Model Learning Agent` complete model search

> `run_complete_qmla()`

`Quantum Hamiltonian Learning` just run the parameter optimisation subroutine. Runs on the model set as `true_model` within the *ES*.

> `run_quantum_hamiltonian_learning()`

`Multi-model quantum Hamiltonian learning` just run the parameter optimisation subroutine. Runs on several models independently; the models are set in the list `qhl_models` within the *ES*.

> `run_quantum_hamiltonian_learning_multiple_models()`

The primary function of the `QuantumModelLearningAgent` class is to manage the model search. Models are assigned a unique `model_ID` upon generation. QMLA considers a set of models as a *layer* or a `BranchQMLA`. Models can reside on multiple branches. For each *ES* included in the instance, an *Exploration Tree (ET)* is built. On a given tree, the associated *ES* determines how to proceed, in particular by deciding which models to consider. The first branch of the tree holds the initial models $\mu^1 = \{M_1^1, \dots M_n^1\}$ for that *ES*. After the initial models have been trained and compared on μ^1, the *ES* uses the available information (e.g. the number of pairwise wins each model has) to construct a new set of models, $\mu^2 = \{M_1^2, \dots M_n^2\}$. Subsequent branches μ^i similarly construct models based on the information available to the *ES* so far.

Each `BranchQMLA` is resident on its associated *ES* tree, but the branch is also known to *QMLA*. Branches are assigned unique IDs by *QMLA*, such that *QMLA* has a birds-eye view of all of the mdoels on all branches on all trees (in general there can be multiple *ES* entertained in a single *instance*). Indeed, a useful way to think of *QMLA* is as a search across a forest consisting of N trees, where each leaf is a unique model, and there can be multiple leaves per branch and multiple branches per tree, with the ultimate goal of identifying the single best leaf for describing the *system*.

When *QMLA* finds that it has completed a layer, it is ready for the next batch of work: it checks whether the *ET* has finished growing, in which case it begins the process of nominating the champion from that *ES*. Otherwise, *QMLA* calls on the *ES* (via the *ET*) to request a set of models, which it places on its next branch, completely indifferent to how those models are generated, or whether they have been learned already. This allows for completely self-contained logic in the *ES*: QMLA will simply learn and compare the models it is presented - it is the responsibility of the *ES* to interpret them. As such, the core *QMLA* algorithm can be thought of as a simple loop: while the *ES* continues to return models, place those models on a branch, learn them and compare them. When all *ES* indicate they are finished,

nominate champions from each *ET*; compare the champions of each tree against each other, and thus determine a *global champion*.

3.2 Exploration Strategy

Exploration Strategies (ES) are the engine of *QMLA*. The *ES* specifies how *QMLA* should proceed at each stage, most importantly by determining the next set of models for *QMLA* to test. These are the primary mechanism by which most users should interface with the *QMLA* framework: by designing an `ExplorationStrategy` which implements the user-specific logic required. In particular, each *ES* must provide a `generate_models()` method to construct models given information about the previous models' training/comparisons. User *ES* classes can be used to specify parameters required throughout the *QMLA* protocol. These are all detailed in the `setup` methods of the `ExplorationStrategy` class; users should familiarise themselves with these settings before proceeding.

At minimum, a functional *ES* should look like:

```
class UserExplorationStrategy(qmla.ExplorationStrategy):
    def __init__(
        self,
        exploration_rules,
        true_model=None,
        **kwargs
    ):
        super().__init__(
            exploration_rules=exploration_rules,
            true_model=true_model,
            **kwargs
        )
        self.true_model = 'pauliSet_1_x_d1+pauliSet_1_y_d1'
```

An example of *ES* design, including a simple greedy-addition model generation method as well as seeting several parameter settings, is:

```
from qmla.shared_functionality import experiment_design_heuristics as edh

class UserExplorationStrategy(qmla.ExplorationStrategy):
    def __init__(
        self,
        exploration_rules,
        true_model=None,
        **kwargs
    ):
        super().__init__(
            exploration_rules=exploration_rules,
            true_model=true_model,
            **kwargs
        )
        # Overwrite true model
        self.true_model = 'pauliSet_1_x_d1+pauliSet_1_y_d1'

        # Overwrite modular functionality
        self.model_heuristic_subroutine = edh.VolumeAdaptiveParticleGuessHeuristic

        # Overwrite parameters
        self.max_num_qubits = 2
        self.num_probes = 10
```

(continues on next page)

(continued from previous page)

```
        self.qinfer_resampler_a = 0.95

        # User specific attributes (not available by default in QMLA)
        self.model_base_terms = [
            "pauliSet_1_x_d2",
            "pauliSet_1_y_d2",
            "pauliSet_1_z_d2",
            "pauliSet_2_x_d2",
            "pauliSet_2_y_d2",
            "pauliSet_2_z_d2",
        ]
        self.search_exhausted = False


    def generate_models(
        self,
        model_list,
        **kwargs
    ):
        if self.spawn_stage[-1] == None:
            # Use spawn_stage for easy signals between calls to this method
            # e.g. to alter the functionality after some condition is method

            self.spawn_stage.append("one_parameter_models")
            return self.model_base_terms

        previous_champion = model_list[0]
        champion_terms = previous_champion.split("+")
        nonpresent_terms = list(set(self.model_base_terms) - set(champion_terms))
        new_models = [
            "{}+{}".format(previous_champion, term) for term in nonpresent_terms
        ]

        if len(new_models) == 1:
            # After this, there will be no more to test,
            # so signal to QMLA that this ES is finished.
            self.search_exhausted = True

        return new_models

    def check_tree_completed(
        self,
        spawn_step,
        **kwargs
    ):
        r"""
        QMLA asks the exploration tree whether it has finished growing;
        the exploration tree queries the exploration strategy through this method.
        """
        return self.search_exhausted
```

In order to implement a new *ES*, *QMLA* searches in the directory `qmla/exploration_strategies`, so the user's *ES* must be `import` ed to the `qmla/exploration_strategies/__init__.py`. *QMLA* retrieves the *ES* through calls to the function *`get_exploration_class()`*, by searching for the *ES* specified in the *Launch* script. For example, the launch script (e.g. at `qmla/launch/local_launch.sh`) should be updated to call the user's *ES*, e.g.

```
#!/bin/bash

###############
# QMLA run configuration
###############
num_instances=1
run_qhl=0
experiments=500
particles=2000

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

exploration_strategy='UserExplorationStrategy'
```

A complete step-by-step example of implementing custom *ES* is given in section_tutorial. Users should ensure they understand the options for launching *QMLA* as outlined in *Launch*.

Each *ES* is assigned a unique *Exploration Tree (ET)*, although most users need not alter the infrastructure of the *ET* or *QMLA*.

3.3 Models

3.3.1 Construction

Models are specified by a string of terms separated by +, e.g. `pauliSet_1_x_d1+pauliSet_1_y_d1`. Model names are unique and are assigned a `model_id` upon generation within *QuantumModelLearningAgent* : *QMLA* will recognise if a model string has already been proposed and therefore been assigned a `model_id`, rather than retraining models which is computationally expensive. The uniqueness of models is ensured by the terms being sorted alphabetically internally within the string (e.g. `pauliSet_1_x_d1+pauliSet_1_y_d1` instead of `pauliSet_1_y_d1+pauliSet_1_x_d1`), but note *QMLA* ensures this internally so users do not need to enfore it in their *generate_models()*.

The strings are processed into models as follows. By separating models into their terms (`model_name.split('+')`), the cardinality (number of terms, n) is found. An $n-$ dimensional Gaussian is constructed to represent the parameter distribution for the model; individual parameters can be specified in `gaussian_prior_means_and_widths` of `_setup_model_learning()`. The terms are then processed into matrices. A number of *String to matrix processing* functions are available by default; new processing functions can be added by the user but must be incorporated in *process_basic_operator()* so that *QMLA* will know where to find them.

3.3.2 Classes

Models are central to the *QMLA* framework so it sensible to identify their core functionality so we can design software to facilitate them. In particular, there are three forms of classes which each depict models, but fulfil different roles. In brief, these classes are

ModelInstanceForLearning Class used for the training of individual models.

ModelInstanceForComparison Class used for comparing models which have already been trained

ModelInstanceForStorage Class retained by *QuantumModelLearningAgent*, storing the results of the model's training and comparisons.

We next detail each of these roles of the model concept.

3.3.3 Training

QMLA relies on a subroutine for training individual candidate models: it is imperative that a given candidate is optimised against the *system*, as otherwise it might appear as a relatively weak candidate compared with its potential. In principle, any parameter learning subroutine can fulfil this role in *QMLA*, such as Hamiltonian tomography or using neural networks for parameter estimation. The in-built facility for this subroutine is *quantum Hamiltonian Learning (QHL)*. We do not descibe the *QHL* protocol here but readers can refer to [WGFC13a], [WGFC13b] for details.

`ModelInstanceForLearning` is a disposable class which instatiates independently from `QuantumModelLearningAgent`. It trains a given model via `qmla.remote_learn_model_parameters()`, performs analysis on the trained model, summarises the outcome of the training and sends a concise data packet to the database, before being deleted. The model training refers to quantum Hamiltonian learning, performed in conjunction with [QInfer], via `update_model()`. Importantly, *QMLA* trains models simply by calling `qmla.remote_learn_model_parameters()`: this function acts emph{remotely} and is therefore independent, allowing for multiple instance of the function and `ModelInstanceForLearning` to run simultaneously. As such, this class mechanism allows for emph{parallel processing} within *QMLA*, enabling speedup proportional to the number of processes available (for the model training stages).

3.3.4 Comparisons

Like the training subroutine, in principle *QMLA* can operate with any model comparison subroutine, but in practice we use *Bayes factors (BF)*. This is a quantity which is used to distinguish between models.

`ModelInstanceForComparison` is a disposable class which reads the redis database to retrieve information about the trainng of the given `model_id`. It then reconstructs the model, e.g. based on the final estimated mean of the parameter distribution. Then, to compare models, `remote_bayes_factor_calculation()` interfaces two instances of `ModelInstanceForComparison` such that each model is exposed to the opponent's experiments for further updates, such that the two models under consideration have identical experiment records (at least partially whereupon the BF is based), allowing for meaningful comparison among the two. This is achieved through `update_log_likelihood()`.

Similiar to the training stage, `remote_bayes_factor_calculation()` can be run in parallel to provide a large speedup to the overall *QMLA* protocol.

3.3.5 Storage

Finally, `ModelInstanceForStorage` is a much smaller onject than the previous forms of the model, which retains only the useful information for storage/analysis within the bigger picture in `QuantumModelLearningAgent`. It retrieves the succinct summaries of the training/comparisons pertainng to a single model which are stored on the redis database, allowing for later anlaysis as required by *QMLA*. The retrieval of trained model data is performed in `model_update_learned_values()`.

3.4 Modular functionality

A large amount of the design of an *ES* involves implementation of subroutines: there are a number of methods of `ExplorationStrategy` which can be overwritten in order to achieve functionality specific to the target *system*. In this section we describe these subroutines. Many of the subroutines have a number of sensible implementations: we make *QMLA* emph{modular} by providing a set of pre-built subroutines, and allow them to be easily swapped so that a new *ES* can benefit from arbitrary combiniations of subroutines. The subroutines are called by wrapper methods in `ExplorationStrategy`; to set which function is called, change the attribute in the definition of the custom *ES*. Alternatively, directly overwrite the wrapper. The pre-built functions are in `qmla/shared_functionality`.

Within `ExplorationStrategy`, these modular functions are set in `_setup_modular_subroutines()`.

An example of setting each of these subroutines is

```
from qmla.shared_functionality import experiment_design_heuristics as edh
from qmla.shared_functionality import expectation_value_functions as ev
from qmla.shared_functionality import \
    qmla.shared_functionality.probe_set_generation as probes
from qmla.shared_functionality import qinfer_model_interface as qii
from qmla.shared_functionality import prior_distributions
from qmla.shared_functionality import latex_model_names as lm

class UserExplorationStrategy(qmla.ExplorationStrategy):
    def __init__(
        self,
        exploration_rules,
        true_model=None,
        **kwargs
    ):
        super().__init__(
            exploration_rules=exploration_rules,
            true_model=true_model,
            **kwargs
        )
        # Overwrite true model
        self.true_model = 'pauliSet_1_x_d1+pauliSet_1_y_d1'

        # Overwrite expectation value subroutine
        self.expectation_value_subroutine = ev.default_expectation_value

        # Overwrite probe generation subroutines
        self.system_probes_generation_subroutine = probes.plus_probes_dict
        self.plot_probes_generation_subroutine = probes.zero_state_probes

        # Overwrite exeperiment design heuristic
        self.model_heuristic_subroutine = edh.VolumeAdaptiveParticleGuessHeuristic

        # Overwrite QInfer interface
        self.qinfer_model_subroutine = qii.QInferModelQMLA

        # Overwrite prior distribution subroutine
        self.prior_distribution_subroutine = priors.gaussian_prior

        # Overwrite latex mapping subroutine
        self.latex_string_map_subroutine = lm.lattice_set_grouped_pauli
```

3.4.1 Probes

The *probe* is the input state used during the learning procedure. Different probes permit different biases on the information available to the algorithm; it is essential to consider which probes are appropriate for learning different classes of models. In general the training procedure loops over the available probes, to minimise the chance of favouring some models due to bias inherent in the probe. For example, if the probe is (close to) an eigenstate of one candidate model, that model will never learn effectively since there will be little variation in measurements correspdonding to evolving the probe according to that model. Intuitively, the most informative probe for a given model is a superposition of its eigenstates, since any evolution in this basis will be reflected by the measurement.

The default set of probes is to use a random set. Alternative sets include $|+\rangle^{\otimes N}$ or $|0\rangle^{\otimes N}$. Probes are generated in a dictionary, of which the keys are `(probe_id, num_qubits)`; `probe_id` runs from 1 to the `num_probes` attribute of the *`ExplorationStrategy`* controls; the `num_qubits` runs from 1 to `max_num_qubits`.

There are a number of sets of probes required, all similarly set by specifying the subroutine:

`system_probes_generation_subroutine` Probes used for evolution on the target *system*

`simulator_probes_generation_subroutine` Probes correspdonding exactly to those used on the system. These should be the same so that the likelihood function is meaningful, but in realistic cases there may be slight differences in probe preparation, e.g. due to expected noise in an experimental system. Therefore it is possible to specify a different set. Note to enable this functionality, `shared_probes` must also be set to `False`.

`plot_probes_generation_subroutine` Probes used for plots throughout the protocol. Plots should be in the same basis for consistency; we generate them once per *run* to save time, since the plot probes are the same everywhere. The standard plotting probes are $|+\rangle^{\otimes N}$.

`evaluation_probe_generation_subroutine` Some *ES* use evaluation datasets within model selection; to specify a different generator than `system_probes_generation_subroutine`, set this attribute. Defaults to `None`.

3.4.2 Experiment design heuristic

In order for the model *Training* to perform well, it is essential that the parameter learning subroutine is fed useful, meaningful data. We use an *experiment design heuristic (EDH)* to generate informative experiments. The *EDH* can encompass custom logic for particular use cases, although the most common (particle guess heuristic [WGFC13a]) attempts to select an evolution time t which can distinguish between strong and weak parameterisations (particles) based on the current distribution.

Primarily the *EDH* must choose an evolution time t and *probe*, since these two together specify an entire experiment in most use cases. *QMLA* can consider more complex experiment designs, in which case the *EDH* must also choose informative values for all inputs.

3.4.3 QInfer interface

As mentioned, the workhorse of model *Training* is [QInfer]. The default behaviour of *`QInferModelQMLA`* is to call *`likelihood()`* for both the calculation of the datum from the *system*, and the likelihoods of all the particles through the simulator. This too can be replaced, for example if calls to the *system* need to interface with a real experiment, or the particles should be computed through a quantum simulator.

3.4.4 Prior distribution

QInfer works by taking an initial prior distribution, which it iteratively narrows based on quantum likelihood estimation. This process of narrowing the distribution is what we call emph{learning}: after N_E experiments worth of data, the mean of the remaining distribution is considered as the optimised parameterisation.

The prior can be altered to incorporate the user's prior knowledge about the system. The default generator for the prior is to construct an n dimensional Gaussian through *gaussian_prior()*. Importantly, the range of each term's parameter can be different, e.g. near-neighbour couplings having much higher frequency than distant neighbours. Terms' prior mean and width can be specified in `gaussian_prior_means_and_widths`. Terms which do not have specific means/widths in `gaussian_prior_means_and_widths` are assigned based on the *ExplorationStrategy* attributes `min_param`, `max_param`: the defaults are

```
mean = (max_param + min_param)/2;
```

```
width= (max_param - min_param)/4.
```

To overwrite this, e.g. to change the default width of each parameter's distribution, users can implement a new prior generation function to replace `prior_distribution_subroutine`.

```
self.gaussian_prior_means_and_widths = {
    'pauliSet_1_x_d1' : (5, 1),
    'pauliSet_1_y_d1' : (150, 25),
    'pauliSet_1_z_d1' : (1e6, 1e2)
}
self.min_param = 0
self.max_param = 10

self.prior_distribution_subroutine = alternatve_prior_generation
```

3.4.5 Latex name mapping

Each model string format requires a method which can map the string to a Latex string. This is because much of the analysis automatically generated by *QMLA* refers to individual models or terms, so it is useful that these can be rendered into a readable format, rather than the raw string used to generate the matrices used by the algorithm. The mapping function should be able to operate either on single terms or entire models strings. If using terms like `pauliSet_i_t_dN`, the default *pauli_set_latex_name()* should work. Further examples, specific to models of bespoke *ES* are *grouped_pauli_terms()*, *fermi_hubbard_latex()*.

```
>>> from qmla.shared_functionality.latex_model_names import grouped_pauli_terms
>>> self.latex_string_map_subroutine = grouped_pauli_terms
```

3.5 Output and Analysis

When a run is launched (either locally or remotely), a results directory is built for that run. In that directory, results are stored in several formats from each instance.

By default, *QMLA* provides a set of analyses, generating several plots in the sub-directories of the run's results directory.

Analyses are available on various levels:

Run results across a number of instances.

Example: the number of instance wins for champion models.

Example: average dynamics reproduced by champion models.

Instance Performance of a single insance.

Example: models generated and the branches on which they reside

Model Individual model performance within an instance.

Example: parameter estimation through QHL.

Example: pairwise comparison between models.

Comparisons Pairwise comparison of models' performance.

Example: dynamics of both candidates (with respect to a single basis).

Within the *Launch* scripts, there is a `plot_level` variable which informs *QMLA* of how many plots to produce by default. This gives users a level of control over how much analysis is performed. For instance, while testing an Exploration Strategy, a higher degree of testing may be required, so plots relating to every individual model are desired. For large runs, however, where a large number of models are generated/compared, plotting each model's training performance is overly cumbersome and is unneccessary.

The plots generated at each plot level are:

```
plot_level=1
    _plot_model_terms()
plot_level=2
plot_level=3
    _plot_dynamics_all_models_on_branches()
    _plot_evaluation_normalisation_records()
plot_level=4
    _plot_learning_summary()
    _plot_dynamics()
    _plot_preliminary_preparation()
    plot_dynamics_from_models()
plot_level=5
    _plot_distributions()
    plot_heuristic_attributes()
plot_level=6
```

3.6 Launch

There are two mechanisms for launching *QMLA*: locally and in parallel. Both of are available through `bash` scripts in `qmla/launch`. When launched in parallel, the model training/comparison subroutines are run on remote processes, e.g. in a compute cluster. In either case, the user has a set of top-level controls, bearing in mind that the majority of user requirements are implemented in the *ExplorationStrategy*. Following the setting of these controls, the remainder of the launch script call a number of `bash` and `Python` scripts for the actual implementation, which most users should not need to alter.

The available controls to the user are

num_instances number of instance in the run

run_qhl if 1, only implements *QHL* on the `true_model` attribute of the *ES*, i.e. *`run_quantum_hamiltonian_learning()`*.

run_qhl_mulit_model if 1, only implements *QHL* on the `qhl_models` attribute (list) of the *ES*, i.e. *`run_quantum_hamiltonian_learning_multiple_models()`*. if *both* this and :run_qhl: are 0, then the full *QMLA* protocol is run (*`run_complete_qmla()`*).

exp Number of experiments used during model training

prt Number of particles used during model training

plot_level specifies the granularity of plots generated. See *Output and Analysis*

debug_mode (bool) whether to run *QMLA* in degug mode. Should not be required by most users; this mode merely logs further data in the instances' log files, which can be found in the *run results directory*.

exploration_strategy specify the name of the *ES* class to use.

alt_exploration_strategies list of alternative *ES* for the case where multiple *ET* s are considered. This list should be in brackets with elements separated by spaces (i.e no commas). Note that in `parallel_launch.sh`, this must be enabled through setting `multiple_exploration_strategies=1`, while in `local_launch.sh` it is sufficient that the list is not empty.

An example of the top few lines of `local_launch.sh` is then given by

```
#!/bin/bash

###############
# QMLA run configuration
###############
num_instances=100
run_qhl=0 # perform QHL on known (true) model
run_qhl_mulit_model=0 # perform QHL for defined list of models.
exp=500 # number of experiments
prt=2000 # number of particles

###############
# QMLA settings - user
###############
plot_level=4
debug_mode=0

###############
# QMLA settings - default
###############
do_further_qhl=0
q_id=0
use_rq=0
further_qhl_factor=1
further_qhl_num_runs=$num_instances
plots=0
number_best_models_further_qhl=5

###############
# Choose exploration strategy/strategies
```

(continues on next page)

(continued from previous page)

```
###############

exploration_strategy='UserExplorationStrategy'

alt_exploration_strategies=(
    'IsingLatticeSet'
    'Genetic'
)
```

3.6.1 Redis server

QMLA uses a redis server as a database and job broker for the implementation of remote tasks. This is launhed automatically when using `parallel_launch.sh`, but using `local_launch.sh`, must be initiated in terminal as

CHAPTER

FOUR

API REFERENCE

4.1 Quantum Model Learning Agent

4.1.1 Manager class

The overall *QMLA* protocol is managed by this class.

class qmla.**QuantumModelLearningAgent**(*qmla_controls=None*, *model_priors=None*, *experimental_measurements=None*, ***kwargs*)

QMLA manager class.

Controls the infrastructure which determines which models are learned and compared. By interpreting user defined *ExplorationStrategy*, grows *ExplorationTree* objects which hold numerous models on *BranchQMLA* objects. All models on branches are learned and then compared. The comparisons on a branch inform the next set of models generated on that tree.

First calls a series of setup functions to implement infrastructure used throughout.

The available algorithms, and their corresponding methods, are:

- Quantum Hamilontian Learning:

  run_quantum_hamiltonian_learning()

- Quantum Hamilontian Learning multiple models:

  run_quantum_hamiltonian_learning_multiple_models()

- Quantum Model Learning Agent:

  run_complete_qmla()

Parameters

- **qmla_controls** (ControlsQMLA) – Storage for configuration of a QMLA instance.
- **model_priors** (*dict*) – values of means/widths to enfore on given models, specifically for further_qhl mode.
- **experimental_measurements** (*dict*) – expectation values by time of the underlying true/target model.

_check_model_exists(*model_name*)

True if model already exists; False if not.

_compile_and_store_qmla_info_summary()

Gather info needed to run QMLA tasks and store remotely.

QMLA issues jobs to run remotely, namely for model (parameter) learning and model comparisons (Bayes factors). These jobs don't need access to all QMLA data, but do need some common info, e.g. number of particles and epochs. This function gathers all relevant information in a single dict, and stores it on the redis server which all worker nodes have access to. It also stores the probe sets required for the same tasks.

_compute_base_resources()
Compute the set of minimal resources for models to learn on.

In the case self.reallocate_resources==True, models will receive resources (epochs, particles) scaled by how complicated they are. For instance, models with 4 parameters will receive twice as many particles as a model with 2 parameters.

_consider_new_model(*model_name*)
Check whether a proposed model already exists.

Check whether the new model *name*, exists in all previously considered models, held in *model_lists*, organised by dimension of models. If name has not been previously considered, 'New' is returned. If name has been previously considered, the corresponding location

in db is returned.

Parameters

- **model_lists** (*dict*) – lists of models already considered, organised by the number of qubits of those models
- **name** (*str*) – model for consideration

_delete_unpicklable_attributes()
Remove elements of QMLA which cannot be pickled, which cause errors if retained.

_fundamental_settings()
Basic settings, path definitions etc.

_get_model_data_by_field(*name*, *field*)
Get any data from the model database corresponding to a given model name.

Parameters

- **name** (*str*) – model name to get data of
- **field** (*str*) – field name to get data corresponding to model

_inspect_remote_job_crashes()
Check if any job on redis queue has failed.

_plot_bayes_factors()
Plot Bayes factors between pairs of models, both by model IDs and by their F-scores.

_plot_branch_champions_dynamics(*all_models=False*, *model_ids=None*)
Plot reproduced dynamics of all branch champions

Parameters

- **all_models** (*bool*) – whether to plot all models in the instance
- **model_ids** (*list*) – list of model IDs to plot dynamics of
- **save_to_file** (*str*) – path at which to save the resultant figure

_plot_branch_champs_quadratic_losses()
Wrapper for `plot_quadratic_loss()`.

_plot_branch_champs_volumes(*model_id_list=None*, *branch_champions=True*, *branch_id=None*, *save_to_file=None*)
Plot the volume of each branch champion within this instance.

Parameters

- **model_id_list** (*list*) – list of model IDs to plot volumes of, if None plot branch champions
- **branch_champions** (*bool*) – force plot only branch champions' volumes
- **branch_id** (*int*) – if provided, plot the volumes of all models within that branch
- **save_to_file** (*str*) – path at which to store the resultant figure.

_plot_dynamics_all_models_on_branches(*branches=None*)
Plot the dynamics of all models on given branches.

Parameters **branches** (*list, optional*) – list of branches to draw dynamics for, defaults to None, in which case all branches are drawn.

_plot_evaluation_normalisation_records()
Plot the normalisation record of all models grouped by the branch they are on.

_plot_exploration_tree(*modlist=None*, *only_adjacent_branches=True*, *save_to_file=None*)
Wrapper for `plot_qmla_single_instance_tree()`

_plot_model_terms(*colour_by='binary'*)
Plot the terms of each model by model ID.

Parameters **colour_by** (*str, optional*) – defaults to 'binary' for black/white; alternatively colour by f_score of model

_plot_one_qubit_probes_bloch_sphere(*save=False*)
Show all one qubit probes on Bloch sphere.

_plot_parameter_learning_champion()
Plot parameter estimates vs experiment number for a single model.

Wrapper for `plot_parameter_estimates()` :param bool true_model: whether to force only plotting the true

model's parameter estimeates

_plot_parameter_learning_single_model(*model_id=0*, *true_model=False*, *save_to_file=None*)
Plot parameter estimates vs experiment number for a single model.

Wrapper for `plot_parameter_estimates()` :param bool true_model: whether to force only plotting the true

model's parameter estimeates

_plot_parameter_learning_true()
Plot parameter estimates vs experiment number for a single model.

Wrapper for `plot_parameter_estimates()` :param bool true_model: whether to force only plotting the true

model's parameter estimeates

_plot_qmla_radar_scores(*modlist=None*, *save_to_file=None*)
deprecated Wrapper for `plotRadar()`.

_plot_r_squared_by_epoch_for_model_list(*modlist=None*, *save_to_file=None*)
Plot R^2 vs experiment number for given model list.

_plot_volume_after_qhl(*model_id=None*, *true_model=True*, *show_resamplings=True*, *save_to_file=None*)
Plot volume vs experiment number of a single model. Wrapper for `plot_volume_after_qhl()`

_potentially_redundant_setup()
Graveyard for deprecated ifnrastructure.

Attributes etc stored here which are not functionally used within QMLA, but which are called somewhere, and cause errors when omitted. Should be stored here temporarily during development, and removed entirely when sure they are not needed.

_set_learning_and_comparison_parameters(*model_priors*, *experimental_measurements*)
Parameters related to learning/comparing models.

_setup_parallel_requirements()
Infrastructure for use when QMLA run in parallel.

_setup_tree_and_exploration_strategies()
Set up infrastructure.

_true_model_definition()
Information related to true (target) model.

_update_database_model_info()
Calls `model_update_learned_values()` for all models learned in this instance.

add_model_to_database(*model*, *exploration_tree*, *branch_id=- 1*, *force_create_model=False*)
Considers adding a model to QMLA's database of models.

Checks whether the nominated model is already present; if not generates a model instance and stores pertinent details in the model database.

Parameters

- **model** (*str*) – name of model to consider
- **branch_id** (*float*) – branch id to associate this model with, if the model is new.
- **force_create_model** (*bool*) – True: add model even if the name is found already. False: (default) check if the model exists before adding

Return dict add_model_output *is_new_model* : bool, whether model is new (True) or has already been added (False) model_id: unique model ID for the model, whether new or existing

analyse_instance()
Basic analysis of this instance

check_champion_reducibility()
Potentially remove negligible terms from the champion model.

Consider whether the champion model has some terms whose parameters were found to be negligible (either within one standard deviation from 0, or very close to zero as determined by the exploration strategy's *learned_param_limit_for_negligibility* attribute). Construct a new model which is the same as the champion, less those negligible terms, named the reduced champion. The data of the champion model is inherited by the reduced candidate model, i.e. its parameter estimates, as well as its history of parameter learning for those which are not negligible. A new *normalization_record* is started, which is used in the comparison between the champion and the reduced champion. Compare the champion with the reduced champion; if the reduced champion is heavily favoured, directly select it as the global champion. This method is triggered if the exploration strategy's *check_champion_reducibility* attribute is set to True.

compare_model_pair(*model_a_id*, *model_b_id*, *return_job=False*, *branch_id=None*, *remote=True*, *wait_on_result=False*)
Launch the comparison between two models.

Either locally or by passing to a job queue, run *remote_bayes_factor_calculation()* for a pair of models specified by their IDs.

Parameters

- **model_a_id** (*int*) – unique ID of one model of the pair
- **model_b_id** (*int*) – unique ID of other model of the pair
- **return_job** (*bool*) – True - return the rq job object from this function call. False (default) - return nothing.
- **branch_id** (*int*) – unique branch ID, if this model pair are on the same branch
- **remote** (*bool*) – whether to run the job remotely or locally True - job is placed on queue for RQ worker False - function is computed locally immediately
- **wait_on_result** (*bool*) – whether to wait for the outcome or proceed after sending the job to the queue.

Returns bayes_factor the Bayes factor calculated between the two models, i.e. BF(m1,m2) where m1 is the lower model id. Only returned when *wait_on_result==True*.

compare_model_set(*model_id_list=None*, *pair_list=None*, *remote=True*, *wait_on_result=False*, *recompute=False*)

Launch pairwise model comparison for a set of models.

If *pair_list* is specified, those pairs are compared; otherwise all pairs within *model_id_list* are compared.

Pairs are sent to *compare_model_pair()* to be computed either locally or on a job queue.

Parameters

- **model_id_list** (*list*) – list of model names to compute comparisons between
- **pair_list** (*list*) – list of tuples specifying model IDs to compare
- **remote** (*bool*) – passed directly to *compare_model_pair()*
- **wait_on_results** (*bool*) – passed directly to *compare_model_pair()*
- **recompute** (*bool*) – whether to force comparison even if a pair has been compared previously

compare_models_within_branch(*branch_id*, *pair_list=None*, *remote=True*, *recompute=False*)

Launch pairwise model comparison for all models on a branch.

If *pair_list* is specified, those pairs are compared; otherwise pairs are retrieved from the *pairs_to_compare* attribute of the branch, which is usually all-to-all.

Pairs are sent to *compare_model_pair()* to be computed either locally or on a job queue.

Parameters

- **branch_id** – unique ID of the branch within the QMLA environment
- **pair_list** (*list*) – list of tuples specifying model IDs to compare
- **remote** (*bool*) – passed directly to *compare_model_pair()*
- **wait_on_results** (*bool*) – passed directly to *compare_model_pair()*
- **recompute** (*bool*) – whether to force comparison even if a pair has been compared previously

compare_nominated_champions()
Compare the champions of all exploration strategy trees.

Get the champions (usually one, but in general can be multiple) from each tree, where each tree is unique to an exploration strategy. Place the champions on a branch together and perform all-versus-all comparisons. The champion of that branch is deemed the global champion.

compute_model_f_score(*model_id*, *model_name=None*, *model_constructor=None*, *exploration_class=None*, *beta=1*)
Compte and store f-score of given model.

Parameters

- **model_id** (*int*) – model ID to compute f-score of
- **beta** (*float*) – for generalised F_beta score. (default) 1 for F1 score.

Return float f_score F-score of given model.

compute_statistical_metrics_by_generation()
Compute, store and plot various statistical metrics of all studied models.

Parameters **save_to_file** (*str*) – path to save the resultant figure in.

finalise_qmla()
Steps to end QMLA algorithm, such as storing analytics.

get_model_storage_instance_by_id(*model_id*)
Get the unique *ModelInstanceForLearning* for the given model_id.

Parameters **model_id** (*int*) – unique ID of desired model

Returns storage class of the model

Return type *ModelInstanceForLearning*

get_results_dict(*model_id=None*)
Store the useful information of a given model, usually the champion.

Parameters **model_id** (*int*) – unique ID of the model whose information to store

Return dict results_dict data which will be stored in the results_{ID}.p file following QMLA's completion.

learn_model(*model_name*, *branch_id*, *blocking=False*)
Learn a given model by calling the standalone model learning functionality.

The model is learned by launching a job either locally or to the job queue. Model learning is implemented by *remote_learn_model_parameters()*, which takes a unique model name (string) and distills the terms to learn. If running locally, QMLA core info is passed. Else if RQ workers are being used, it retrieves QMLA info from the shared redis database, and the function is launched via rq's *Queue.enqueue* function. This puts a task on the redis *Queue* - the task is the implementation of *remote_learn_model_parameters()*. The effect is either to learn the model here, or else to have launched a job where it will be learned remotely, so nothing is returned.

Parameters

- **model_name** (*str*) – string uniquely representing a model
- **branch_id** (*int*) – unique branch ID within QMLA environment
- **use_rq** (*bool*) – whether to use RQ workers, or implement locally
- **blocking** (*bool*) – whether to wait on model to finish learning before proceeding.

learn_models_on_given_branch(*branch_id*, *blocking=False*)
Launches jobs to learn all models on the specified branch.

Models which are on the branch but have already been learned are not re-learned. For each remaining model on the branch, *learn_model()* is called. The branch is added to the redis database *active_branches_learning_models*, indicating that branch_id has currently got models in the learning phase. This redis database is monitored by the *learn_models_until_trees_complete()*. When all models registered on the branch have completed, it is recorded, allowing QMLA to perform the next stage: either spawning a new branch from this branch, or continuing to the final stage of QMLA. This method can block, meaning it waits for a model's learning to complete before proceeding. If in parallel, do not block as model learning won't be launched until the previous model has completed.

Parameters

- **branch_id** (*int*) – unique QMLA branch ID to learn models of.
- **use_rq** (*bool*) – whether to implement learning via RQ workers. Argument only used when passed to *QuantumModelLearningAgent.learn_model()*.
- **blocking** (*bool*) – whether to wait on all models' learning before proceeding.

learn_models_until_trees_complete()
Iteratively learn/compare/generate models on exploration strategy trees.

Each *ExplorationStrategy* has a unique `QMLATree`. Trees hold sets of models on `BranchTree` objects.

Models on a each branch are learned through *learn_models_on_given_branch()*. Any model which has previously been considered defaults to the earlier instance of that model, rather than repeating the calculation. When all models on a branch are learned, they are all compared through *compare_models_within_branch()*.

When a branch has completed learning and comparisons of models, the corresponding tree is checked to see if it has finished proposing models, through *is_tree_complete()*. If the tree is not complete, the *next_layer()* method is called to generate the next branch on that tree. The next branch can correspond to *spawn* or *prune* stages of the tree's *ExplorationStrategy*, but QMLA is ambivalent to the inner workings of the tree/exploration strategy: a branch is simply a set of models to learn and compare.

When all trees have completed learning, this method terminates.

log_print(*to_print_list*)
Wrapper for *print_to_log()*

new_branch(*model_list*, *pairs_to_compare='all'*, *pairs_to_compare_by_names=None*, *exploration_strategy=None*, *spawning_branch=0*)
Add a set of models to a new QMLA branch.

Branches have a unique id within QMLA, but belong to a single tree, where each tree corresponds to a single exploration strategy.

Parameters

- **model_list** (*list*) – strings corresponding to models to place in the branch
- **pairs_to_compare** (*str or list*) – set of model pairs to perform comparisons between. 'all' (deafult) means all models in *model_list* are set to compare. Otherwise a list of tuples of model IDs to compare
- **exploration_strategy** (*str*) – exploration strategy identifer; used to get the unique tree object corresponding to an exploration strategy, which is then used to host the branch.

- **spawning_branch** (*int*) – branch id which is the parent of the new branch.

Returns branch id which uniquely identifies the new branch within the QMLA environment.

plot_instance_outcomes()
Generate plots corresponding to this instance.

A number of plotting routines are called, depending on the plot_level set by the user at launch.

process_comparisons_within_branch(*branch_id*, *pair_list=None*)
Process comparisons between models on the same branch.

(Similar functionality to *process_model_set_comparisons()*, but additionally updates some branch infrastructure, such as updating the branch's *champion_id*, *bayes_points* attributes). Pairwise comparisons are retrieved and processed by *process_model_pair_comparison()*, which informs the superior model. For each pairwise comparison a given model wins, it receives a single point. All comparisons are weighted evenly. Model points are gathered; the model with most points is deemed the champion of the set. If a subset of models have the same (highest) number of points, that subset is compared directly, with the nominated champion deemed the champion of the wider set.

Parameters **branch_id** (*int*) – unique ID of the branch whose models to compare

Returns

- models_points: the points (number of comparisons won) of each model on the branch
- champ_id: unique model ID of the champion model within the set

process_model_pair_comparison(*a=None*, *b=None*, *pair=None*)
Process a comparison between two models.

The comparison (Bayes factor) result is retrieved from the redis database and used to update data on the models.

Parameters

- **a** (*int*) – one of the model's unique ID
- **b** (*int*) – one of the model's unique ID
- **pair** (*tuple*) – alternative mechanism to provide the model IDs, effectively (a,b)

Returns ID of the model which is deemed superior from this pair

process_model_set_comparisons(*model_list*)
Process comparisons between a set of models.

Pairwise comparisons are retrieved and processed by *process_model_pair_comparison()*, which informs the superior model.

For each pairwise comparison a given model wins, it receives a single point.

All comparisons are weighted evenly. Model points are gathered; the model with most points is deemed the champion of the set.

If a subset of models have the same (highest) number of points, that subset is compared directly, with the nominated champion deemed the champion of the wider set.

Parameters **model_list** (*list*) – list of model names to compete

Returns unique model ID of the champion model within the set

run_complete_qmla()
Run complete Quantum Model Learning Agent algorithm.

Each *ExplorationStrategy* is assigned a QMLATree, which manages the exploration strategy. When new models are spawned by an exploration strategy, they are placed on a BranchQMLA of the corresponding tree. Models are learned/compared/spawned iteratively in *learn_models_until_trees_complete()*, until all trees declare that their exploration strategy has completed. Exploration Strategies are complete when they have nominated one or more champions, which can follow spawning/pruning stages as required by the exploration strategy. Nominated champions are then compared with *compare_nominated_champions()*, resulting in a single global champion selected. Some analysis then takes place, including possibly reducing the selected global champion if it is found that some of its terms are not impactful.

run_quantum_hamiltonian_learning()
Run Quantum Hamiltonian Learning algorithm .

The *true_model* of the *ExplorationStrategy* is used to generate true data (in simulation) and have its parameters learned.

run_quantum_hamiltonian_learning_multiple_models(*model_names=None*)
Run Quantum Hamiltonian Learning algorithm with multiple simulated models.

Numerous Hamiltonian models attempt to learn the dynamics of the true model. The underlying model is set in the *ExplorationStrategy*'s *true_model* attribute.

Parameters **model_names** (*list*) – list of strings of model names to learn the parameterisations of. None: taken from *ExplorationStrategy* *qhl_models*.

spawn_from_branch(*branch_id*)
Retrieve the next set of models and place on a new branch.

By checking the QMLATree` associated with the *branch_id* used to call this method, call *ExplorationTree.next_layer()*, which returns a set of models to place on a new branch, as well as which models therein to compare. These are passed to *new_branch()*, constructing a new branch in the QMLA environment. The generated new branch then has all its models learned by calling *learn_models_on_given_branch()*. *next_layer()* is in control of how to select the next set of models, usually either by calling the *ExplorationStrategy*'s *generate_models()* or *tree_pruning()* methods. This allows the user to define how models are generated, given access to the comparisons of the previous branch, or how the tree is pruned, e.g. by performing preliminary parent/child branch champion comparisons.

Parameters **branch_id** (*int*) – unique ID of the branch which has completed

store_bayes_factors_to_csv(*save_to_file*, *names_ids='latex'*)
deprecated Store the pairwise comparisons computed during this instance. model_bayes_factorsCSV() removed and is needed TODO if wanted, find in old github commits and reimplement.

Wrapper for model_bayes_factorsCSV().

store_bayes_factors_to_shared_csv(*bayes_csv*)
Store the pairwise comparisons computed during this instance in a CSV shared by all concurrent instances.

4.2 Logistics

Here we list some of the functionality used as the logistics to implement *QMLA*.

4.2.1 User controls

ControlsQMLA Controls (user and otherwise) to specify QMLA instance.

class qmla.**ControlsQMLA**(*arguments*, ***kwargs*)
Storage for configuration of a QMLA instance.

Command line arguments specify details about the QMLA instance, such as number of experiments/particles etc, required to implement the QMLA instance. The command line arguments are stored together in this class. The class is then given to the *qmla.QuantumModelLearningAgent* instance, which uses those details into the implementation. Some QMLA parameters are also set by the attributes of the Exploration Strategy. In particular, the *ExplorationStrategy* of the true model is instantiated by calling *get_exploration_class()*. model is defined as the true model of that instance. This exploration strategy instance is the master exploration strategy for the QMLA instance: the true Likewise, instances are generated for all of the exploration strategies specified by the user: these instances are associated with the exploration strategy *ExplorationTree* objects.

Parameters **arguments** (*dict*) – command line arguments, parsed into a dict.

log_print(*to_print_list*)
Wrapper for *print_to_log()*

4.2.2 Database framework

Operator Object for mathematical properties of a single model.

Functions

qmla.**get_num_qubits**(*name*)
Parse string and determine number of qubits this operator acts on.

Default convention is to use a naming mechanism specified by string_processing_functions(). In all such constructions, the final element of each term is *dN*, so we can extract the number of qubits N.

If using old convention where terms are tensor-producted by T, TT, TTT... , we find the largest T string instance, from which we deduce the number of qubits. - xTx = pauli_x TENSOR_PRODUCT pauli_x –> 2 qubits - yTyTTy = pauliy_y TENSOR_PRODUCT pauli_y TENSOR_PRODUCT pauli_y –> N=3 i.e. the largest tensor product of of length is N-1.

Parameters **name** (*str*) – name of model

qmla.**get_constituent_names_from_name**(*name*)
Separate into separate terms in model name. e.g. 'pauliSet_1_x_d2+pauliSet_1_y_d2' -> ['pauliSet_1_x_d2', 'pauliSet_1_y_d2'] :param str name: name of model

qmla.**alph**(*name*)
Alphabetise the model name.

If name newer follows convention where terms are separated by +, simply separate them. If name follows older convention, analyse to separate terms and then alphabetise them. Parse string and recursively call alph function to alphabetise substrings.

Parameters **name** (*str*) – name of model to alphabetise

4.2.3 Model Generation

qmla.**process_basic_operator**(*basic_operator*)

Transform a string, representing a term in the model, into a matrix.

Physical systems have different corresponding string processing functions. These are provided in the dictionary *qmla.process_string_to_matrix.string_processing_functions*. There are a number of rules which model strings must obey to be processed properly.

- Terms are separated by +.
- Within terms, components are separated by _.
- Components have different meanings, depending on which string processing function is used.
- The first component is the *indicator* of which processing function to use; it is matched with a processing function in *~qmla.string_processing_functions*.
- The final component in general indicates the dimension N of the system, and is specified by dN.
- No other component should start with d, as it uniquely indicates the dimension.
- Alternatively, core operators can be processed alone, these are given in core_operator_dict.

For example, the string pauliSet_1J2_xJx_d3+pauliSet_1J3_zJz_d3:

- Terms: pauliSet_1J2_xJx_d3, pauliSet_1J3_zJz_d3
- Components (of pauliSet_1J2_xJx_d3): pauliSet, 1J2, xJx, d3
- Indicator pauliSet tells it to process via process_multipauli_term().
- d3 tells it to use a 3 qubit basis
- Other components are interpreted by the string processing function
- In this case, the result is the matrix (XXI + ZIZ) .

Parameters **basic_operator** (*str*) – term to generate matrix from.

Return np.ndarray mtx matrix corresponding to the input term.

4.2.4 String to matrix processing

These functions map strings to matrices which can be used in the construction of models.

4.2.5 Initialising Exploration Strategy

qmla.**get_exploration_class**(*exploration_rules*, ***kwargs*)

Get an instance of the class specified by the user which implements an exploration Strategy.

Instance of a ExplorationStrategy (or subclass). This is used to specify how QMLA proceeds, in particular by designing the next batch of models to test. Exploration Strategy is specified by the name passed to implement_qmla in the launch script, through the command line flag *exploration_strategy*. This string is searched for in the exploration_classes dictionary. New exploration strategies must be added here so that QMLA can find them.

Parameters **exploration_rules** (*str*) – string corresponding to an exploration strategy

Params **kwargskwargs** arguments required by the exploration strategy, passed directly to the desired exploration strategy's constructor.

Return ExplorationStrategy gr exploration strategy class instance

4.2.6 Trees and branches

class qmla.**ExplorationTree**(*exploration_class*)

Tree corresponding to an exploration strategy for management within QMLA.

Each *ExplorationStrategy* in the *QuantumModelLearningAgent* instance is associated with one tree (instance of this class). The tree interacts with the QMLA environment, for instance to choose the next set of models to learn through *next_layer()*. *BranchQMLA* exist both on the tree and the QMLA environment.

Parameters **exploration_class** (ExplorationStrategy) – instance of the exploration strategy to associate with this tree.

finalise_tree(***kwargs*)

After learning/pruning complete, run finalise_model_learning().

get_initial_models()

Models for the first layer of this exploration strategy.

Return list initial_models list of model names to place on the first layer corresponding to this ES

Return list pairs_to_compare list of model name pairs to compare

is_tree_complete()

Check if tree has completed and doesn't need any further branches.

log_print(*to_print_list*)

Wrapper for *print_to_log()*

new_branch_on_tree(*branch_id*, *models*, *pairs_to_compare*, *model_storage_instances*, *precomputed_models*, *spawning_branch*, ***kwargs*)

Add a branch to this tree.

Generate a new *BranchQMLA*, and places the given models on it, and assign it to this tree. Return the object, so it can also act as a branch/layer in the *QuantumModelLearningAgent* environment. Note models can reside on multiple branches.

Parameters

- **branch_id** (*int*) – branch ID unique to this branch and new in the QMLA environment.
- **models** (*list*) – model names to place on this branch.
- **pairs_to_compare** (*list*) – list of pairs of models to compare on this branch.
- **model_storage_instances** (*dict*) – *ModelInstanceForStorage* instances of each model id to be placed on this branch.
- **precomputed_models** (*list*) – models to place on branch which have already been learned on a previous branch.
- **spawning_branch** (*int*) – branch ID spawning this branch, to be recorded as the new branch's parent.

Return BranchQMLA branch instance of branch

next_layer(***kwargs*)

Determine the next set of models, for the next branch of this exploration strategy tree.

These are generated iteratively in stages, until that stage is marked as complete within the exploration strategy. * Spwaning: using the set of models from the previous branch to construct new models,

via *generate_models()*.

- The spawning stage is terminated when *check_tree_completed()* returns True. For example, if a predetermined number of spawn steps have passed, or if the model learning has converged.
- After the spawn stage, the tree enters the pruning stage. For example, parental collapse: branch champions of this tree are compared with their parents; if either parent or child is strongly superior, the loser does not progress to the next branch.
- Pruning is carried out by *tree_pruning()*, until :meth:`~qmla.exploration_strategies.ExplorationStrategy.check_tre returns True.
- After the pruning stage, it must be possible for this tree to nominate a list of models to be considered for global champion, by `nominate_champion()`.

The set of models decided by this method are placed on the next QMLA branch. `pairs_to_compare` here specifies which pairs of models on that next branch should be compared. Typically, model learning corresponds to spawning; during this stage all models on branches should be compared. Pruning then corresponds to comparing models which have already been leared (though new unlearned models will be learned). During pruning, it may not be necessary to compare all pairs, for instance if it is desired only to compare parent/child pairs at a given branch.

Parameters **kwargs** (*dict*) – key word args passed directly to spawning/pruning functions.

Return list model_list list of model names to place on the next QMLA branch

Return str or list pairs_to_compare which model IDs within the model_list to perform comparisons between.

nominate_champions()

After tree has completed, nominate champion(s) to contest for global champion.

Exploration Strategies can nominate one or more models as global champion. The nominated champions of all ESs enter an all-vs-all contest in *compare_nominated_champions()*, with the winner deemed the global champion. This tree object interfaces with the ES, so here it is just a wrapper for `nominate_champions()`.

class qmla.**BranchQMLA**(*branch_id*, *models*, *model_storage_instances*, *pairs_to_compare*, *tree*, *precomputed_models*, *spawning_branch*)

update_branch(*pair_list*, *models_points=None*)

Process results for this branch.

Parameters

- **pair_list** (*list*) – pairs of models which were compared on this branch
- **models_points** (*dict*) – results of comparisons

4.2.7 Parameter definition

qmla.**set_shared_parameters**(*exploration_class*, *run_info_file=None*, *all_exploration_strategies=[]*, *run_directory=''*, *num_particles=100*, *probe_max_num_qubits_all_exploration_strategies=12*, *generate_evaluation_experiments=True*)

Set up parameters for this *run* of QMLA. A run consists of any number of independent QMLA instances; for consistency they must share the same information. Parameters, such as true model (system) parameters and probes to use for plotting purposes, are shared by all QMLA instances within a given run.

This function does not return anything, but stores data required for the run to the `run_info_file` path. The data pickled as `run_info` are:

True_model name of true model, i.e. the model we call the system, against which candidate models are tested

Params_list list of parameters of the true model

Params_dict dict of parameters of the true model

Exploration_rule exploration strategy (name) of true model

All_exploration_strategies list of all exploration strategies (names) which are to be performed by each instance

Evaluation_probes proebs to use during evaluation experiments

Evaluation_times times to use during evaluation experiments

Parameters

- **exploration_class** (*ExplorationStrategy*) – exploration strategy of true model, from which to extract key info, e.g. true parameter ranges and prior.
- **run_info_file** (*str*) – path to which to store system information
- **all_exploration_strategies** (*list*) – list of instances of *ExplorationStrategy* which are the alternative exploration strategies, i.e. which are performed during each instance, but which do not specify the true model (system).
- **run_directory** (*str*) – path to which all results/information pertaining to this unique QMLA run are stored
- **num_paritlces** (*int*) – number of particles used during model learning
- **probe_max_num_qubits_all_exploration_strategies** (*int*) – largest system size for which to generate plot probes
- **generate_evaluation_experiments** (*bool*) – whether to construct an evaluation dataset which can be used to objectively evaluate models. Evaluation data consists of experiments (i.e. probes and evolution times) which were not typically used in model learning, therefore each model can be compared fairly on this data set.

4.2.8 Redis

qmla.**get_redis_databases_by_qmla_id**(*host_name*, *port_number*, *qmla_id*, *tree_identifiers=None*)

Gets the set of redis databases unique to this QMLA instance.

Each *QuantumModelLearningAgent* instance is associated with a unique redis database. Redis databases are specified by their *hostname* and *port number*. All workers for the QMLA instance can read the redis databsae of that instance. Data required by various workers is stored here, through *_compile_and_store_qmla_info_summary()*.

A set of databases are stored at the redis database host_name:port_number; these are listed in `qmla.redis_settings.databases_required`.

Parameters

- **host_name** (*str*) – name of host server on which redis database exists.
- **port_number** (*int*) – this QMLA instance's unique port number (6300 + qmla_id).
- **qmla_id** (*int*) – QMLA id, unique to a single instance within a run.

Return dict database_dict set of database addresses unique to the qmla_id, host_name and port_number.

qmla.**get_seed**(*host_name*, *port_number*, *qmla_id*)
Unique seed for this QMLA id.

Numerous databases can belong to a given host:port address, and these are identified by their `db` attribute (a number to keep databases separate). Databases are seeded using the `qmla_id`, as well as the host:port, to avoid multiple QMLA instances, which can exist on the same host and port, clashing and interfering with each others' data.

E.g. * A host:port is already in use for `qmla_id=1`, which uses a set of 5 databases. * `qmla_id=2` requests a set of databases on the same host:port * The first available `db` is 6, such that `qmla_id=2` will not interfere with

the databases of `qmla_id=1`.

Parameters

- **host_name** (*str*) – name of host server on which redis database exists.
- **port_number** (*int*) – this QMLA instance's unique port number (6300 + qmla_id).
- **qmla_id** (*int*) – QMLA id, unique to a single instance within a run.

Return int seed unique number to use as the starting `db` for a given QMLA instances set of databases.

4.2.9 Logging

qmla.**print_to_log**(*to_print_list*, *log_file*, *log_identifier=''*)
Writes to the log file, registering the time and identifier.

Adds the content of *to_print_list* to the *log_file*, using the *log_identifier* to indicate where a given log entry originated.

Parameters

- **to_print_list** (*str() or list()*) – string you want to print
- **log_file** (*str()*) – path of the log file you want to update
- **log_identifier** (*str()*) – identifier for the log

4.3 Models

4.3.1 Model for training

Model to perform parameter learning upon, usually *QHL*.

This is a disposable class which instatiates indepdendently from *QuantumModelLearningAgent*, trains the model via *qmla.remote_learn_model_parameters()*, performs analysis on the trained model, summarises the outcome of the training and sends a concise data packet to the database, before being deleted. The model training refers to *QHL*, performed in conjunction with [QInfer], via *update_model()*.

class qmla.**ModelInstanceForLearning**(*model_id*, *model_name*, *qid*, *exploration_rule*, *log_file*, *qmla_core_info_database=None*, *host_name='localhost'*, *port_number=6379*, ***kwargs*)
Model used for parameter learning.

Models are specified by their name; they can be separated into separate terms by splitting the name string by '+'. Individual terms correspond to base matrices and are assigned parameters. Each term is assigned a parameter probability distribution, or a prior distribution: this will be iteratively changed according to evidence from experiments, and its mean gives the estimate for that parameter. Prior distributions are used by the QInfer updater, and can be specified by the *get_prior()* method. The individual terms are parsed into matrices for calculations. This is achieved by *process_basic_operator()*: different string syntax enable different core oeprators.

Parameter estimation is done by *update_model()*. The final parameter estimates are set as the mean of the posterior distribution after n_experiments wherein n_particles are sampled per experiment (these user definted parameters are retrieved from *qmla_core_info_dict*). *learned_info_dict()* returns the pertinent learned information.

Parameters

- **model_id** (*int*) – ID of the model to study
- **model_name** (*str*) – name of the model to be learned
- **qid** – ID of the QMLA instance
- **exploration_rule** (*str*) – name of exploration_strategy
- **qmla_core_info_database** (*dict*) – essential details about the QMLA instance needed to learn/compare models. If None, this is retrieved instead from the redis database.
- **host_name** (*str*) – name of host server on which redis database exists.
- **port_number** (*int*) – port number unique to this QMLA instance on redis database
- **log_file** (*str*) – path of QMLA instance's log file.

_consider_reallocate_resources()
Model might get less resources if it is deemed less complex than others

_finalise_learning()
Record and log final result.

_initialise_model_for_learning(*model_name*, *qmla_core_info_database*, ***kwargs*)
Preliminary set up necessary before parameter learning.

Start instances of classes used throughout, generally by calling the exploration strategy's method,

- qinfer inferface: `qinfer_model()`.
- updater is default QInfer.SMCUpdater.
- parameter distribution prior: *get_prior()*.

Parameters

- **model_name** (*str*) – name of the model to be learned
- **exploration_rule** (*str*) – name of exploration_strategy
- **qmla_core_info_database** (*dict*) – essential details about the QMLA instance needed to learn/compare models. If None, this is retrieved instead from the redis database.

_initialise_tracking_infrastructure()
Arrays, dictionaries etc for tracking learning across experiments

_model_plots_old()
Generate plots specific to this model. Which plots are drawn depends on the `plot_level` set in the launch script.

_plot_distributions()
For each parameter, plot: * prior distribution * posterior distributino * prior distribution for comparison, i.e. posterior from learning recast as a unimodal normal * true parameters (if applicable) * learned parameter estimates * covariance matrix between parameters (separate plot)

TODO add plotting levels: run, instance, model

_plot_dynamics()
Plots the dynamics reproduced by this model against system data.

_plot_learning_summary()

Plot summary of this model's learning:

- parameter estimates and uncertainties
- volume of parameter distribution
- experimental times used
- (resample points superposed on the above)
- likelihoods of system/particles
- difference between system/particles' likelihoods

_plot_posterior_mesh_pairwise()
Plots the posterior mesh as contours for each pair of parameters.

Mesh from qinfer.SMCUpdater.posterior_mesh

_plot_preliminary_preparation()
Prepare model for plots; make directory.

_record_experiment_updates(*update_step*, *new_experiment=None*, *datum=None*, *update_time=0*)
Update tracking infrastructure.

_setup_qinfer_infrastructure()
Set up prior, model and updater (via QInfer) which are used to run Bayesian inference.

_store_prior()
Save the prior raw and as plot.

compute_likelihood_after_parameter_learning()
" Evaluate the model after parameter learning on independent evaluation data.

learned_info_dict()
Place essential information after learning has occured into a dict.

This is used to recreate the model for

- comparisons: *ModelInstanceForComparison*
- storage within the main QMLA environment `ModelInstanceForStorage>`.

log_print(*to_print_list*, *log_identifier=None*)
Wrapper for *print_to_log()*

log_print_debug(*to_print_list*)
Log print if global debug_log_print set to True.

update_model()
Run updates on model, corresponding to quantum Hamiltonian learning procedure.

This function is called on an instance of this model to run the entire QHL algorithm.

Get datum corresponding to true system, where true system is either experimental or simulated, by calling simulate_experiment on the QInfer.SMCUpdater. This datum is taken as the true expected value for the system, which is used in the likelihood calucation in the Bayesian inference step. This is done by calling the *update* method on the qinfer_updater. Effects of the update are then recorded by `_record_experiment_updates()`, and terminate either upon convergence or after a fixed *num_experiments*. Final details are recorded by `_finalise_learning()`.

4.3.2 Model for comparisons

Model to use during Bayes factor comparisons.

This is a disposable class which reads the redis database to retrieve information about the trainng of the given model ID. It then reconstructs the model, e.g. based on the final estimated mean of the parameter distribution. Then, it is interfaced with a competing instance of the class within `remote_bayes_factor_calculation()`: the opponent's experiments are used for further updates to the present model, such that the two models under consideration have identical experiment records (at least partially whereupon the BF is based), allowing for meaningful comparison among the two.

class qmla.**ModelInstanceForComparison**(*model_id*, *qid*, *opponent*, *qmla_core_info_database=None*, *learned_model_info=None*, *host_name='localhost'*, *port_number=6379*, *log_file='QMD_log.log'*)

Model instances used for Bayes factor comparisons.

When Bayes factors are calculated remotely (ie on RQ workers), they require infrastructure to do calculations, e.g. QInfer SMCUpdater instances. This class captures the minimum required to enable these calculations. After learning, important data from `ModelInstanceForLearning` is stored on the redis database. This class unpickles the useful information and generates new instances of the updater etc. to use in the comparison calculations.

If run locally, *qmla_core_info_database* and *learned_model_info* can be passed directly to this class, to save unpickling data from the redis database.

Parameters

- **model_id** (*int*) – ID of the model to study
- **qid** – ID of the QMLA instance
- **qmla_core_info_database** (*dict*) – essential details about the QMLA instance needed to learn/compare models. If None, this is retrieved instead from the redis database.
- **learned_model_info** (*dict*) – result of learning, generated by `learned_info_dict()`.
- **host_name** (*str*) – name of host server on which redis database exists.
- **port_number** (*int*) – port number unique to this QMLA instance on redis database
- **log_file** (*str*) – path of QMLA instance's log file.

log_print(*to_print_list*, *log_identifier=None*)

Wrapper for `print_to_log()`

log_print_debug(*to_print_list*)

Log print if global debug_log_print set to True.

plot_dynamics(*ax*, *times*)

Plot dynamics of this model after its parameter learning stage.

Parameters

- **ax** – matplotlib axis to plot on
- **times** (*list*) – times against which to plot

update_log_likelihood(*new_times*, *new_experimental_params*)

4.3.3 Model for storage

This object is much smaller than the other forms of the model, i.e. those used for training (*ModelInstanceForLearning*) and comparisons (*ModelInstanceForComparison*), which retains only the useful information for storage/analysis within the bigger picture in *QuantumModelLearningAgent*. It retrieves the succinct summaries of the training/comparisons pertainng to a single model which are stored on the redis database, allowing for later anlaysis as required by *QMLA*.

class qmla.**ModelInstanceForStorage**(*model_name*, *model_id*, *qid*, *plot_probes=None*, *qmla_core_info_database=None*, *host_name='localhost'*, *port_number=6379*, *log_file='QMD_log.log'*, ***kwargs*)

Model stored in QMLA environment.

Retrieves data after model is trained remotely, so that *qmla.QuantumModelLearningAgent* can access that data.

Parameters

- **model_name** (*str*) – name of model under study
- **model_id** (*int*) – ID of model which is unique to QMLA instance
- **model_terms_matrices** (*np.array()*) – list of matrices corresponding to the operators which compose the model
- **plot_probes** (*dict*) – probes used in all plots for consistency
- **qmla_core_info_database** (*dict*) – essential details about the QMLA instance needed to learn/compare models. If None, this is retrieved instead from the redis database.
- **host_name** (*str*) – name of host server on which redis database exists.
- **port_number** (*int*) – port number unique to this QMLA instance on redis database
- **log_file** (*str*) – path of QMLA instance's log file.

compute_expectation_values(*times=[]*)

Get the expectation values using the learned Hamiltonian.

Construct Hamiltonian from estimated learned parameters, and compute the expectation values, using the same input state as used for plotting. Stores a dictionary of { t : expectation value }.

Parameters **times** (*list*) – times to use

log_print(*to_print_list*)

Wrapper for *print_to_log()*

model_update_learned_values(*learned_info=None*, ***kwargs*)

Get result of model learning and store within this object.

Every element stored by *learned_info_dict()* is stored as an attribute here.

Parameters **learned_info** (*dict*) – results of remote model learning if None, retrieved from the redis database if not None, computed locally and passed

r_squared(*times=None*, *min_time=0*, *max_time=None*)

Compute and store r squared for given times.

Parameters

- **times** (*list*) – times to use for calculation
- **min_time** (*float*) – minimum time to use for calculation
- **min_time** – maximum time to use for calculation

Return float final_r_squared r squared of the learned model against the times given

r_squared_by_epoch(*times=None*, *min_time=0*, *max_time=None*, *num_points=10*)

Compute and store r squared up to all times.

TODO incorporate as flag in r_squared() to store by epoch instead of separate fnc.

4.4 Implementation

4.4.1 Model learning

qmla.**remote_learn_model_parameters**(*name*, *model_id*, *branch_id*, *exploration_rule*, *qmla_core_info_dict=None*, *remote=False*, *host_name='localhost'*, *port_number=6379*, *qid=0*, *log_file='rq_output.log'*)

Standalone function to perform Quantum Hamiltonian Learning on individual models.

Used in conjunction with redis databases so this calculation can be performed without any knowledge of the QMLA instance.

Given model ids and names are used to instantiate the ModelInstanceForLearning class, which is then used for learning the models parameters.

QMLA info is unpickled from a redis databse, containing true operator, params etc.

Once parameters are learned, we pickle the results to dictionaries held on a redis database which can be accessed by other actors.

Parameters

- **name** (*str*) – model name string
- **model_id** (*int*) – unique model id
- **branch_id** (*int*) – QMLA branch where the model was generated
- **exploration_rule** (*str*) – string corresponding to a unique exploration strategy, used by get_exploration_class to generate a ExplorationStrategy (or subclass) instance.
- **qmla_core_info_dict** (*dict*) – crucial data for QMLA, such as number of experiments/particles etc. Default None: core info is stored on the redis database so can be retrieved there on a server; if running locally, can be passed to save pickling.
- **remote** (*bool*) – whether QMLA is running remotely via RQ workers.
- **host_name** (*str*) – name of host server on which redis database exists.
- **port_number** (*int*) – this QMLA instance's unique port number, on which redis database exists.
- **qid** (*int*) – QMLA id, unique to a single instance within a run. Used to identify the redis database corresponding to this instance.
- **log_file** (*str*) – Path of the log file.

4.4.2 Model comparison

qmla.**remote_bayes_factor_calculation**(*model_a_id*, *model_b_id*, *branch_id=None*, *bf_data_folder=None*, *times_record='BayesFactorsTimes.txt'*, *check_db=False*, *bayes_threshold=1*, *host_name='localhost'*, *port_number=6379*, *qid=0*, *log_file='rq_output.log'*)

Standalone function to compute Bayes factors.

Used in conjunction with redis databases so this calculation can be performed without any knowledge other than model IDs. Data is unpickled from a redis databse, containing *learned_model* information, i.e. final parameters etc. Given `model_id`s correspond to model names in the database, which are combined with the final learned parameters to reconstruct model classes of complete learned models. Each model had been trained on a given set of experimental parmaeters (times). The reconstructed model classes are updated according to the experimental parameters of the opponent model, such that both models have underwent the same experiments. From these we extract log likelihoods to compute the Bayes factor, BF(A,B). Models have a unique pair_id, simply (min(A,B), max(A,B)). For BF(A,B) >> 1, A is deemed the winner; BF(A,B)<<1 deems B the winner. The result is then stored redis databases:

- bayes_factors_db: BF(A,B)
- bayes_factors_winners_db: id of winning model
- active_branches_bayes: when complete, increase the count of complete pairs' BF on the given branch.

Parameters

- **model_a_id** (*int*) – unique id for model A
- **model_b_id** (*int*) – unique id for model B
- **branch_id** (*int*) – unique id of branch the pair (A,B) are on
- **or int num_times_to_use** (*str*) – how many times, used during the training of models A,B, to use during the BF calculation. Default 'all'; if otherwise, Nt, keeps the most recent Nt experiment times of A,B.
- **bf_data_folder** (*str*) – folder path to store information such as times used during calculation, and plots of posterior marginals.
- **times_record** (*str*) – filename to store times used during calculation.
- **check_db** (*bool*) – look in redis databases to check if this pair's BF has already been computed; return pre-computed BF if so.
- **bayes_threshold** (*float*) – value to determine whether either model is superior enough to "win" the comparison. If 1 < BF < threshold, neither win.
- **host_name** (*str*) – name of host server on which redis database exists.
- **port_number** (*int*) – this QMLA instance's unique port number, on which redis database exists.
- **qid** (*int*) – QMLA id, unique to a single instance within a run. Used to identify the redis database corresponding to this instance.
- **log_file** (*str*) – Path of the log file.

qmla.remote_bayes_factor.**plot_dynamics_from_models**(*models*, *exp_msmts*, *bf_times*, *bayes_factor*, *save_directory*, *figure_format='png'*)

Plot the dynamics of the pair of models considered in a Bayes factor comparison.

Parameters

- **models** (*ModelInstanceForLearning*) – list of 2 models which were compared during this calculation, [model_a, model_b].
- **exp_msmts** (*dict*) – times and expectation values for the system.
- **bf_times** (*list*) – Times used for the BF calculation
- **bayes_factor** (*float*) – Bayes factor between the two input models, to be read as BF(model_a, model_b)
- **save_directory** (*path*) – path where the generated figure is to be saved

4.5 Exploration Strategies

ExplorationStrategy Exploration Strategies drive the progression of the *QMLA* algorithm, as described in *Exploration Strategy*.

class qmla.exploration_strategies.**ExplorationStrategy**(*exploration_rules*, *true_model=None*, ***kwargs*)

User defined mechanism to control which models are considered by QMLA.

By changing the attributes, various aspects of QMLA are altered. A number of exploration strategy attributes point to standalone methods available within QMLA, e.g. to generate probes according to a desired mechanism. This allows the user to easily change functionality in a modular fashion. To develop a new exploration strategy, users should read the definitions of all exploration strategy attributes listed in the various `setup` methods, and ensure that the default are suitable for their system, or that they have replaced them in their custom exploration strategy. The `setup` methods are:

- `_setup_modular_subroutines()`
- `_setup_true_model()`
- `_setup_model_learning()`
- `_setup_tree_infrastructure()`
- `_setup_logistics()`

check_tree_completed(*spawn_step*, ***kwargs*)

QMLA asks the exploration tree whether it has finished growing; the exploration tree queries the exploration strategy through this method

exploration_strategy_finalise()

Steps needed to finalise the exploration strategy.

gaussian_prior(*model_name*, *default_sigma=None*, ***kwargs*)

Genearates a QInfer Gaussian distribution .

Given a model_name, deteremines the number of terms in the model, N. Generates a multivariate distribution with N dimensions. This is then used as the initial prior, which QHL uses to learn the model parameters. By default, each parameter's mean is the average of param_min and param_max, with sigma = mean/4. This can be changed by specifying prior_specific_terms:

individual parameter's means/sigmas can be given.

Parameters

- **model_name** (*str*) – Unique string representing a model.
- **param_minimum** (*float*) – Lower bound for distribution.
- **param_maximum** (*float*) – Upper bound for distribution.
- **default_sigma** (*float*) – Width of distribution desired. If None, defaults to 0.25 * (param_max - param_min).
- **prior_specific_terms** (*dict*) – Individual parameter mean and sigma to enforce in the distribution.
- **log_file** (*str*) – Path of the log file for logging errors.
- **log_identifier** (*str*) – Unique identifying sting for logging.

Return QInfer.Distribution dist distribution to be used as prior for parameter learning of the named model.

generate_models (*model_list*, ***kwargs*)

Determine the next set of models for this exploration strategy.

This method is the main driver of QMLA. This method is called iteratively during the `spawn` stage of QMLA, until *check_tree_completed()* returns `True`, for instance after a fixed depth of spawning. In particular it is called by *next_layer()*, which either spawns on the ES tree, or prunes it.

Custom ESs must use this method to determine a set of models for QMLA to consider on the next layer (or *BranchQMLA*) of QMLA. Such a set of models can be constructed based on the results of the previous layers, or according to any logic required by the ES.

Custom methods to replace this have access to the following parameters, and must return the same format of outputs. # TODO remove old/unused data passed to this method

Parameters

- **model_list** (*list*) – list of models on the previous QMLA layer, ordered by their ranking on that layer.
- **model_names_ids** (*dict*) – map `ID : model_name` for all models in the *QuantumModelLearningAgent* instance.
- **called_by_branch** (*int*) – the branch ID from which QMLA is spawning. This does not always need to be set; it is mostly used by the *ExplorationTree* to track which models/branches are parents/children of each other.
- **branch_model_points** (*dict*) – `` ID : number_wins `` number of wins of each model in the previous branch.
- **evaluation_log_likelihoods** (*dict*) – `` ID : eval_log_likel `` foe each model in the previous branch, where `eval_log_likel` is the log likelihood computed against a set of validation data (i.e. not the data on which the model was trained.)
- **model_dict** (*dict*) – lists of models in the QMLA instance, organised by corresponding number of qubits.

Return list model_names names of models as unique strings where terms in each model are separated by +, and each term in each model is interpretable by *process_basic_operator()*.

generate_plot_probes (*probe_maximum_number_qubits=None*, ***kwargs*)

Call the ES's `plot_probes_generation_subroutine`.

Generates a set of probes against which to compute measurements for plotting purposes. The same probe dict is used by all QMLA instances within a run for consistency.

Plot probe generation methods must adhere to the same rules as in *generate_probes()*.

Parameters **probe_maximum_number_qubits** (*int*) – how many qubits to compose probes up to. Can be left None, in which case assigned based on ES's max_num_qubits, or forced to a different value by passing to function call.

Return dict plot_probe_dict set of states against which all models are plotted over time in dynamics plots.

generate_probes(*probe_maximum_number_qubits=None*, *store_probes=True*, ***kwargs*)

Call the ES's probe generation methods to set the system and simulator probes.

In general it is possible for the system and simulator to have different probe states (e.g. due to noise). These can be generated from the same or different methods. if shared_probes is True, then probe_generation_function is called once and the same probes are used for the system as simulator. else simulator_probes_generation_subroutine is called for the simulator probes.

Probe generation methods must take parameters

max_num_qubits number of qubits to go up to when generating probes

num_probes number of probces to produce

Probe generation methods must return

probe_dict A set of probes with num_probes states for each of 1, ..., N qubits up to max_num_qubits. Probe dictionaries should have keys which are tuples of the number of qubits and a probe ID, i.e. (probe_id, num_qubits).

Parameters

- **probe_maximum_number_qubits** (*int*) – how many qubits to compose probes up to. Can be left None, in which case assigned based on ES's max_num_qubits, or forced to a different value by passing to function call.
- **store_probes** (*bool*) – whether to assign the generated probes to the ES instance. If False, probe dict is just returned .

Returns dict new_probes (if not storing) dictionary of probes returned from probe generation function, fulfilling the requirements outlined above.

get_expectation_value(***kwargs*)

Call the ES's measurement_probability_function to compute quantum likelihood.

Compute the probability of measuring in some basis, to be used as likelihood. The default probability is that of the expectation value. Given an input state $\|\psi\rangle$, $P(\hat{H}, t, \|\psi\rangle) = \|\langle\|e^{-i\hat{H}t}\|\psi\rangle\|^2$. However it is possible to use alternative measurements, for instance corresponding to a physical measurement scheme such as Hahn echo or Ramsey sequences.

Modular functions here must take as parameters

ham Hamiltonian to compute probability of

t time to evolve ham for

state proobe state to compute probability with

****kwargs** any further inputs required can be passed as kwargs

Modular functions must return P : the probability of measurement according to custom requirements, to be used as likelihood in *(interactive) quantum likelihood estimation*.

get_heuristic(***kwargs*)
Call the ES's `model_heuristic_function` to build an experiment design heuristic class.

The heuristic class is called upon to design experiments to perform on the system during model learning.

Heuristics should inherit from `BaseHeuristic`. Details of requirements for custom heuristics can be found in the defintion of `BaseHeuristic`. # TODO clear up - the heuristic is a class, not a function

get_measurements_by_time()
Measure the true model for a series of times.

In some experiment design heuristics, those prescribed times are the only ones available to the learning procedure. Other heuristics allow the choice of any experimental time in principle. In either case, the measurements generated here are computed using the `plot_probes`, which are shared by all QMLA instances within the run. They are used for all dynamics plots.

get_prior(*model_name*, ***kwargs*)
Call the ES's `prior_distribution_subroutine` function.

Parameters **model_name** (*str*) – model for which to construct a prior distribution

Return qinfer.Distribution prior N-dimensional distribution used by QInfer as the starting distribution for learning model parameters.

get_qinfer_model(***kwargs*)
Call the ES's `qinfer_model_class` to build the interface with QInfer used for model learning.

The default QInfer model class, and details of what to include in custom classes, can be found in `QInferModelQMLA`.

get_true_parameters()
Retrieve parameters of the true model and use them to construct the true Hamiltonian.

True parameters are set once per run and shared by all instances within that run. Therefore the true parameters are generated only once by *set_shared_parameters()*, and stored to a file which is accessible by all instances within the run.

This method retrieves those shared true parameters and stores them for use by the *QuantumModelLearningAgent* instance and its subsidiary models and methods. It then uses the true parameters to construct `true_hamiltonian` for the ES.

latex_name(*name*, ***kwargs*)
Call the ES's `latex_string_map_subroutine`.

Map a model name (string) to its LaTeX representation.

Parameters **name** (*str*) – name of model to map.

Return str latex_name representation of input model as LaTeX string.

name_branch_map(*latex_mapping_file*, ***kwargs*)
Assign branch to model for visual representation of ES as tree.

Only used for attempt to plot the QMLA instance as a single tree, which is often not suitable, so this is not essential.

plot_dynamics_of_true_model(*probe_dict*, *times*)
Given a set of probes and times, plot their dynamics.

set_specific_plots()
Over-writeable method to set the target plotting methods. Also place any manual plotting methods in here, i.e. which require arguments.

tree_pruning(*previous_prune_branch*)
Get next model set through pruning.

true_model_latex()
Latex representation of true model.

property true_model_terms
Terms (as latex strings) which make up the true model

In order to initialise an *ES*, *QMLA* calls this function, which searches within the namespace of `qmla/exploration_strategies`.

`qmla.`**get_exploration_class**(*exploration_rules*, ***kwargs*)
Get an instance of the class specified by the user which implements an exploration Strategy.

Instance of a `ExplorationStrategy` (or subclass). This is used to specify how QMLA proceeds, in particular by designing the next batch of models to test. Exploration Strategy is specified by the name passed to implement_qmla in the launch script, through the command line flag *exploration_strategy*. This string is searched for in the exploration_classes dictionary. New exploration strategies must be added here so that QMLA can find them.

Parameters **exploration_rules** (*str*) – string corresponding to an exploration strategy

Params ****kwargs****kwargs** arguments required by the exploration strategy, passed directly to the desired exploration strategy's constructor.

Return ExplorationStrategy gr exploration strategy class instance

4.6 Modular functionality

As outlined in *Modular functionality*, some subroutines are modular; here we list some of the availbale implementations.

4.6.1 Experiment Design Hueristics

class qmla.shared_functionality.experiment_design_heuristics.**ExperimentDesignHueristic**(**arg*
Any,
***kw*
Any)

Experiment Design Heuristic base class, to be inherited by specific implementations. This object has access to the QInfer Updater and Model objects, so it can, e.g., sample from the particle distribution, to use these values in the design of a new experiment.

Parameters

- **updater** (*QInfer Updater object*) – QInfer updater for SMC
- **model_id** (*int*) – ID of model under study, defaults to 1
- **oplist** (*list, optional*) – list of matrices representing the operators constituting this model, defaults to None
- **norm** (*str, optional*) – type of norm to use, defaults to 'Frobenius'

- **inv_field** (*str, optional*) – inversion field to use (legacy - should not matter) defaults to 'x_'
- **t_field** (*str, optional*) – name of field corresponding to t, defaults to 't'
- **maxiters** (*int, optional*) – manimum number of iterations to attempt to find distinct particles from the distribution, defaults to 10
- **other_fields** (*list, optional*) – optional further fields, defaults to None
- **inv_func** (*function, optional*) – inverse function, used by QInfer, (legacy - should not matter) defaults to identity
- **t_func** (*function, optional*) – function for computing t, defaults to identity
- **log_file** (*str, optional*) – path to log file, defaults to 'qmla_log.log'

design_experiment (***kwargs*)
Design an experiment. Children classes can overwrite this function to implement custom logic

for the deisggn of experiments.

finalise_heuristic (***kwargs*)
Any functionality the user wishes to happen at the final call to the heuristic.

log_print (*to_print_list*)
Wrapper for *print_to_log()*

plot_heuristic_attributes (*save_to_file*, ***kwargs*)
Summarise the heuristic used for the model training through several plots.

volume of distribution at each experiment

time designed by heuristic for each experiment

effecitve sample size at each experiment, used to determine when to resample

Parameters **save_to_file** (*path*) – path to which the summary figure is stored

4.6.2 Expectation Values

qmla.shared_functionality.expectation_value_functions.**default_expectation_value**(*ham*, *t*, *state*, *log_file='qmla_*, *log_identifier=*, *Value'*)

Default probability calculation: | <state.transpose | e^{-iHt} | state> |**2

Returns the expectation value computed by evolving the input state with the provided Hamiltonian operator.

Parameters

- **ham** (*np.array*) – Hamiltonian needed for the time-evolution
- **t** (*float*) – Evolution time
- **state** (*np.array*) – Initial state to evolve and measure on
- **log_file** (*str*) – (optional) path of the log file
- **log_identifier** (*str*) – (optional) identifier for the log

Returns probability of measuring the input state after Hamiltonian evolution

4.6.3 Prior probability distributions

qmla.shared_functionality.prior_distributions.**gaussian_prior**(*model_name*, *param_minimum=0*, *param_maximum=1*, *default_sigma=None*, *random_mean=False*, *prior_specific_terms={}*, *log_file='qmd.log'*, *log_identifier=None*, ***kwargs*)

Genearates a QInfer Gaussian distribution .

Given a model_name, deteremines the number of terms in the model, N. Generates a multivariate distribution with N dimensions. This is then used as the initial prior, which QHL uses to learn the model parameters. By default, each parameter's mean is the average of param_min and param_max, with sigma = mean/4. This can be changed by specifying prior_specific_terms:

individual parameter's means/sigmas can be given.

Parameters

- **model_name** (*str*) – Unique string representing a model.
- **param_minimum** (*float*) – Lower bound for distribution.
- **param_maximum** (*float*) – Upper bound for distribution.
- **default_sigma** (*float*) – Width of distribution desired. If None, defaults to 0.25 * (param_max - param_min).
- **prior_specific_terms** (*dict*) – Individual parameter mean and sigma to enforce in the distribution.
- **log_file** (*str*) – Path of the log file for logging errors.
- **log_identifier** (*str*) – Unique identifying sting for logging.

Return QInfer.Distribution dist distribution to be used as prior for parameter learning of the named model.

4.6.4 QInfer Interface

class qmla.shared_functionality.qinfer_model_interface.**QInferModelQMLA**(**args: Any*, ***kwargs: Any*)

Interface between QMLA and QInfer.

QInfer is a library for performing Bayesian inference on quantum data for parameter estimation. It underlies the Quantum Hamiltonian Learning subroutine employed within QMLA. Bayesian inference relies on comparisons likelihoods of the target and candidate system. This class, specified by an exploration strategy, defines how to compute the likelihood for the user's system. Most functionality is inherited from QInfer, but methods listed here are edited for QMLA's needs. The likelihood function given here should suffice for most QMLA

implementations, though users may want to overwrite get_system_pr0_array and get_simulator_pr0_array, for instance to specify which experimental data points to use.

Parameters

- **model_name** (*str*) – Unique string representing a model.
- **modelparams** (*np.ndarray*) – list of parameters to multiply by operators, unused for QMLA reasons but required by QInfer.
- **oplist** (*np.ndarray*) – Set of operators whose sum defines the evolution Hamiltonian (where each operator is associated with a distinct parameter).
- **true_oplist** (*np.ndarray*) – list of operators of the target system, used to construct true hamiltonian.
- **trueparams** (*np.ndarray*) – list of parameters of the target system, used to construct true hamiltonian.
- **num_probes** (*int*) – number of probes available in the probe sets, used to loop through probe set
- **probes_system** (*dict*) – set of probe states to be used during training for the system, indexed by (probe_id, num_qubits).
- **probes_simulator** (*dict*) – set of probe states to be used during training for the simulator, indexed by (probe_id, num_qubits). Usually the same as the system probes, but not always.
- **exploration_rule** (*str*) – string corresponding to a unique exploration strategy, used to generate an **explorationStrategy_** instance.
- **experimental_measurements** (*dict*) – fixed measurements of the target system, indexed by time.
- **experimental_measurement_times** (*list*) – times indexed in experimental_measurements.
- **log_file** (*str*) – Path of log file.

are_models_valid (*modelparams*)

Checks that the proposed models are valid.

Before setting new distribution after resampling, checks that all parameters have same sign as the initial given parameter for that term. Otherwise, redraws the distribution. Modified from qinfer.

property expparams_dtype

Returns the dtype of an experiment parameter array.

For a model with single-parameter control, this will likely be a scalar dtype, such as `"float64"`. More generally, this can be an example of a record type, such as `[('time', py.'float64'), ('axis', 'uint8')]`. This property is assumed by inference engines to be constant for the lifetime of a Model instance. In the context of QMLA the expparams_dtype are assumed to be a list of tuple where the first element of the tuple identifies the parameters (including type) while the second element is the actual type of of the parameter, typicaly a float. (Modified from Qinfer).

get_simulator_pr0_array (*particles*, *times*, *probe*)

Compute pr0 array for the simulator.

For user specific data, or method to compute simulator data, replace this function in exploration_strategy.qinfer_model_subroutine.

Here we pass the candidate model's operators and particles to **default_pr0_from_modelparams_times_**.

Parameters

- **times** (*list*) – times to compute pr0 for; usually single element.
- **particles** (*np.ndarry*) – list of particles (parameter-lists), used to construct Hamiltonians.

Returns np.ndarray pr0 probabilities of measuring specified outcome

get_system_pr0_array(*times*, *probe*)

Compute pr0 array for the system. # TODO compute e^(-iH) once for true Hamiltonian and use that rather than computing every step.

For user specific data, or method to compute system data, replace this function in exploration_strategy.qinfer_model_subroutine.

Parameters **times** (*list*) – times to compute pr0 for; usually single element.

Returns np.ndarray pr0 probabilities of measuring specified outcome on system

likelihood(*outcomes*, *modelparams*, *expparams*)

Function to calculate likelihoods for all the particles

Inherited from Qinfer: Calculates the probability of each given outcome, conditioned on each given model parameter vector and each given experimental control setting.

QMLA modifications: Given a list of experiments to perform, expparams, extract the time list. Typically we use a single experiment (therefore single time) per update. QInfer passes particles as modelparams. QMLA updates its knowledge in two steps:

- "simulate" an experiment (which can include outsourcing from here to perform a real experiment),
- update parameter distribution by comparing Np particles to the experimental result

It is important that the comparison is fair, meaning:

- The evolution time must be the same
- The probe state to evolve must be the same.

To simulate the experiment, we call QInfer's simulate_experiment, which calls likelihood(), passing a single particle. The update function calls simulate_experiment with Np particles. Therefore we know, when a single particle is passed to likelihood, that we want to call the true system (we know the true parameters and operators by the constructor of this class). So, when a single particle is detected, we circumvent QInfer by triggering get_system_pr0_array. Users can overwrite this function as desired; by default it computes true_hamiltonian, and computes the likelhood for the given time. When >1 particles are detected, pr0 is computed by constructing Np candidate Hamiltonians, each corresponding to a single particle, where particles are chosen by Qinfer and given as modelparams. This is done through get_simulator_pr0_array. We know calls to likelihood are coupled: one call for the system, and one for the update, which must use the same probes. Therefore probes are indexed by a probe_id as well as their dimension. We track calls to likelihood() in _a and increment the probe_id to pull every second call, to ensure the same probe_id is used for system and simulator.

Parameters

- **outcomes** (*np.ndarray*) – outcomes of the experiments
- **modelparams** (*np.ndarray*) – values of the model parameters particles A shape `(n_particles, n_modelparams)` array of model parameter vectors describing the hypotheses for which the likelihood function is to be calculated.

- **expparams** (*np.ndarray*) – experimental parameters, A shape `(n_experiments, )` array of experimental control settings, with `dtype` given by `expparams_dtype`, describing the experiments from which the given outcomes were drawn.

Return type np.ndarray

Returns A three-index tensor `L[i, j, k]`, where `i` is the outcome being considered, `j` indexes which vector of model parameters was used, and where `k` indexes which experimental parameters where used. Each element `L[i, j, k]` then corresponds to the likelihood $\Pr(d_i|\boldsymbol{x}_j; e_k)$.

log_print (*to_print_list*, *log_identifier=None*)
Writng to unique QMLA instance log.

log_print_debug (*to_print_list*)
Log print if global debug_mode set to True.

property modelparam_names
Returns the names of the various model parameters admitted by this model, formatted as LaTeX strings. (Inherited from Qinfer)

property n_modelparams
Number of parameters in the specific model typically, in QMLA, we have one parameter per model.

n_outcomes (*expparams*)
Returns an array of dtype `uint` describing the number of outcomes for each experiment specified by `expparams`.

Parameters **expparams** (*numpy.ndarray*) – Array of experimental parameters. This array must be of dtype agreeing with the `expparams_dtype` property.

4.6.5 Latex name mapping

Some examples of working latex name maps are provided.

qmla.shared_functionality.latex_model_names.**pauli_set_latex_name** (*name*, ***kwargs*)
Get latex string for model of Hubbard type.

Individual terms must be of the form, to implement operator `t` on qubit `i` of an `N`-qubit system:

```
pauliSet_i_t_dN
```

```
>>> model_name = 'pauliSet_1_x_d2+pauliSet_2_y_d2'
>>> pauli_set_latex_name(model_name)
```

Parameters **name** (*str*) – name of model or term to map

qmla.shared_functionality.latex_model_names.**grouped_pauli_terms** (*name*, ***kwargs*)

qmla.shared_functionality.latex_model_names.**fermi_hubbard_latex** (*name*, ***kwargs*)
Get latex string for model of Hubbard type.

Parameters **name** (*str*) – name of model or term to map

CHAPTER

FIVE

APPLICATIONS

5.1 NV centre characterisation

The model searches presented in [GFK20] have exploration strategies as presented here.

5.1.1 Greedy search

class qmla.exploration_strategies.nv_centre_spin_characterisation.**NVCentreExperimentalData**

Study experimental data.

Uses the same model generation/comparison strategies as *SimulatedExperimentNVCentre*, but targets data measured from a real system. This is done by using an alternative qinfer_model_subroutine, which searches in the dataset for the system's likelihood, rather than computing it.

get_measurements_by_time()
Uses the experimental data as target system data.

class qmla.exploration_strategies.nv_centre_spin_characterisation.**FullAccessNVCentre**(*explorat*
***kwarg*

Exploration strategy for NV system described in experimental paper, assuming full access to the state so the likelihood is based on :math`langle++ | e^{ -ihat{H(vec{x})} t } | ++ rangle`.

This is the base class for results presented in the experimental paper, namely Fig 2. The same model generation strategy is used in each case (i), (ii), (iii): this ES is for (i) pure simulation.

generate_models(*model_list*, *spawn_step*, *model_dict*, ***kwargs*)
Determine the next set of models for this exploration strategy.

This method is the main driver of QMLA. This method is called iteratively during the `spawn` stage of QMLA, until *check_tree_completed()* returns `True`, for instance after a fixed depth of spawning. In particular it is called by *next_layer()*, which either spawns on the ES tree, or prunes it.

Custom ESs must use this method to determine a set of models for QMLA to consider on the next layer (or *BranchQMLA*) of QMLA. Such a set of models can be constructed based on the results of the previous layers, or according to any logic required by the ES.

Custom methods to replace this have access to the following parameters, and must return the same format of outputs. # TODO remove old/unused data passed to this method

Parameters

- **model_list** (*list*) – list of models on the previous QMLA layer, ordered by their ranking on that layer.

- **model_names_ids** (*dict*) – map `ID : model_name` for all models in the *QuantumModelLearningAgent* instance.
- **called_by_branch** (*int*) – the branch ID from which QMLA is spawning. This does not always need to be set; it is mostly used by the *ExplorationTree* to track which models/branches are parents/children of each other.
- **branch_model_points** (*dict*) – `` ID : number_wins `` number of wins of each model in the previous branch.
- **evaluation_log_likelihoods** (*dict*) – `` ID : eval_log_likel `` foe each model in the previous branch, where `eval_log_likel` is the log likelihood computed against a set of validation data (i.e. not the data on which the model was trained.)
- **model_dict** (*dict*) – lists of models in the QMLA instance, organised by corresponding number of qubits.

Return list model_names names of models as unique strings where terms in each model are separated by +, and each term in each model is interpretable by *process_basic_operator()*.

class qmla.exploration_strategies.nv_centre_spin_characterisation.**SimulatedExperimentNVCent**

Uses all the same functionality, growth etc as *TieredGreedySearchNVCentre*, but uses an expectation value which traces out the environment, mimicing the Hahn echo measurement.

This is used to generate (ii) simulated data in the Nature Physics 2021 paper.

class qmla.exploration_strategies.nv_centre_spin_characterisation.**TieredGreedySearchNVCentr**

Exploration strategy for NV system described in Nature Physics 2021 paper, assuming full access to the state so the likelihood is based on $\langle ++|e^{-iH(\hat{\boldsymbol{x}})t}|++\rangle$.

This is the base class for results presented in the experimental paper, namely Fig 2. The same model generation strategy is used in each case (i), (ii), (iii):

this ES is for (i) pure simulation.

check_tree_completed(*spawn_step*, ***kwargs*)
QMLA asks the exploration tree whether it has finished growing; the exploration tree queries the exploration strategy through this method

generate_models(*model_list*, ***kwargs*)
Overwrites `qmla.QuantumModelLearningAgent.generate_models()`.

Constructs models in tiers, where each tier is explored greedily, and only the strongest model from the tier is progressed as the seed model for the subsequent tier.

5.1.2 Genetic algorithm for spin bath

class qmla.exploration_strategies.nv_centre_spin_characterisation.nature_physics_2021.**NVCen**

Exploration strategy for studying large model space of NV centre through a genetic algorithm.

Model generation is through the genetic algorithm exploration strategy, `Genetic`. This *Exploration Strategy* sets up the true model as an NV centre spin interacting with a number of nuclei, and makes a wider number of nuceli searchable by the genetic algorithm. The NV centre is approximated by the Gali approximation [SCG13]. Candidate models are assumed to have been learned extremely well by a parameter esimation algorithm, which may be unrealistic in some cases. In the genetic algorithm, to assess candidate models, we use an objective

function which computes the average residual between the candidate and the system's dynamics, against a representative dataset.

_get_secular_approx_true_params(*num_qubits=2*, *total_num_qubits=5*)
Using the secular approximation, define true parameters for all present terms.

Parameters

- **num_qubits** (*int*) – number of qubits in the target model
- **total_num_qubits** (*int*) – number of qubits of the search space, i.e. terms will be defined in this dimension, even if the system is not expected to be this large.

Returns dict true_params frequencies of each term to include in the true model

_set_true_params()
Set up the target model: call a series of subroutines to define the true model, as well as setting the parameters to represent the physics appropriately.

_setup_available_terms_gali_model(*n_qubits=2*, *available_axes=['z']*)
Generates the set of terms to include in the genetic algorithm.

Terms are stored as an attribute of the class.

Parameters

- **n_qubits** (*int*) – number of qubits to construct terms up to
- **available_axes** (*lsit*) – axes about which to generate terms, under the Gali approximation

_setup_prior_by_parameters()
Constructs the prior distribution to assign true parameters in the model.

These are set in the gaussian_prior_means_and_widths attribute of this exploration strategy class.

generate_evaluation_data(*num_times=100*, ***kwargs*)
Generates sequential, equally spaced times for evaluating the candidate models against.

Parameters **num_times** (*int*) – number of datapoints to generate

Returns dict eval_data set of experiments for model evaluation

get_evaluation_prior(*model_name*, *estimated_params*, *cov_mt*, ***kwargs*)
Generate a QInfer distribution representing the trained model's paramterisation, in order to evaluate that model.

Parameters

- **model_name** (*str*) – string representing the candidate model
- **estimated_params** (*dict*) – average values of the posterior distribution after training, representing the parameter estimates for the model
- **cov_mt** (*np.array*) – covariance matrix, i.e. the relationship between parameters after training

get_prior(*model_name*, ***kwargs*)
Given a candidate model, constructs a very thin prior.

This is done to skip the model training stage, and assumes the training has performed extremely well. This method is called by QMLA in constructing candidate models.

Parameters **model_name** (*str*) – string representing the model which is being tested.

Returns prior QInfer object, used for sampling parameter values when considering the given model

5.2 Genetic Algorithms

Genetic algorithms can be used within the *Exploration Strategy* of *QMLA*; here we provide a genetic algorithm framework which can be plugged in.

class qmla.shared_functionality.genetic_algorithm.**GeneticAlgorithmQMLA**(*genes*, *num_sites*, *true_model=None*, *base_terms=['x', 'y', 'z']*, *selection_method='roulette'*, *crossover_method='one_poi*
mutation_method='element_wise'
mutation_probability=0.1, *selection_truncation_rate=0.5*, *num_protected_elite_models*
unchanged_elite_num_generati
log_file=None, ***kwargs*)

Standalone genetic algorithm implementation for integration with `qmla.QuantumModelLearningAgent`.

This class works with the `ExplorationStrategy` to construct models according to the genetic strategy.

Parameters

- **genes** (*list*) – individual terms which can be combined to form chromosomes
- **num_sites** (*int*) – maximum dimension permitted in model search
- **true_model** (*str*) – target model. if None, set at random from space of valid models.
- **base_terms** (*list*) – deprecated TODO remove
- **selection_method** (*str*) – mechanism through which to select chromosomes as parents. Currently only 'roulette' available, but the framework should facilitate alternatives.
- **crossover_method** (*str*) – mechanism through which parent chromosomes are combined to form offspring. Currently only 'one_point' available, but the framework should facilitate alternatives.
- **mutation_method** (*str*) – mechanism through which to perform chromosome mutation Currently only 'element_wise' available, but the framework should facilitate alternatives.

- **mutation_probability** (*float*) – rate with which the mutation mechanism incurs mutation.
- **selection_truncation_rate** (*float*) – fraction of models to retain as viable parents to the subsequent generation; the lower-rated other models are discarded.
- **num_protected_elite_models** (*int*) – number of models to automatically admit to the subsequent generation.
- **unchanged_elite_num_generations_cutoff** (*int*) – after this number of generations, if the top model has not changed, the model search is terminated.
- **log_file** (*str*) – path of QMLA instance's log file.

basic_pair_selection(*chromosome_selection_probabilities*, ***kwargs*)
Mechanism for selecting two models from the database of potential parents.

Parameters **chromosome_selection_probabilities** (*pd.DataFrame*) – database indicating the probability that every valid pair of parents should be selected.

Return tuple selected_chromosomes two models

chromosome_f_score(*chromosome*)
Get the F score of a candidate model from its chromosome representation.

Parameters **chromosome** (*np.array*) – representation of candidate model

Returns float f_score F score, between 0 and 1, indicating how many terms overlap between the candidate and target models.

chromosome_string(*c*)
Map a chromosome array to a string.

consolidate_generation(*model_fitnesses*, ***kwargs*)
Following the training of all models on a generation, consolidate that generation.

This involves determining the strongest models from the generation, and constructing the database of parent-pairs and their associated selection probabilities.

Parameters **model_fitnesses** (*dict*) – the fitness of each model in this generation according to the chosen objective function.

crossover(***kwargs*)
Wrapper for crossover mechanism.

This method assumes only 2 chromosomes to crossover and passes them to the method set as self.crossover_method, which can be easily replaced to facilitate alternative crossover schemes.

element_wise_mutation(***kwargs*)
Probabilistically mutate each gene independently.

elite_ranking_top_n_models(*model_fitnesses*, ***kwargs*)
Get the top N models, and store info on the elite models to date.

Parameters **model_fitnesses** (*dict*) – the fitness of each model in this generation according to the chosen objective function.

genetic_algorithm_step(*model_fitnesses*, ***kwargs*)
Perform a complete step of the genetic algorithm, assuming all of the required steps have been performed. That is, the database for parent selection must already be available.

Parameters **model_fitnesses** (*dict*) – the fitness of each model in this generation according to the chosen objective function.

Returns list new_models set of models to place on the next generation.

get_base_chromosome()
Creates basic chromosome, i.e. with all genes set to 0.

get_elite_models(***kwargs*)

Wrapper for elite model selection method, here set to self.elite_ranking_top_n_models.

get_pair_selection_order()
Use the probabilities of parental selection to define the order in which to generate offspring. It is cheaper to perform this once than call the database repeatedly.

Return list pair_selection_order list of tuples of the order in which to pass the model pairs to the crossover mechanism to generate offspring

get_selection_probabilities(***kwargs*)
Wrapper for parent selection function, here set to self.truncate_to_top_half.

log_print(*to_print_list*)
Wrapper for *print_to_log()*

map_chromosome_to_model(*chromosome*)
Given a chromosome, get the corresponding model.

Parameters **chromosome** (*np.array*) – chromosome representing a candidate model

Returns str model_string name of the corresponding model

map_model_to_chromosome(*model*)
Given a model, get the corresponding chromosome.

Parameters **model** (*str*) – name of candidate model

Returns np.array chromosome array of ones and zeros indicating which genes are active in the model

model_f_score(*model_name*)
Get the F score of a candidate model.

Parameters **model_name** (*str*) – name of candidate model

Returns float f_score F score, between 0 and 1, indicating how many terms overlap between the candidate and target models.

mutation(***kwargs*)
Wrapper for mutation mechanism. All input arguments to the mutation method are passed directly to the nominated mutation function, set as self.mutation_method.

one_point_crossover(***kwargs*)
Crossover two chromosomes about a single gene.

Input two chromosomes, and selection (a dict) in kwargs. selection contains `chromosome_1` and `chromosome_2`, as well as a dict called `other_data` containing `cut`, which is the position about which to crossover the two chromosomes.

prepare_chromosome_pair_dataframe(*chromosome_probabilities*, *force_mutation=False*)
Given a set of individual chromosome fitnesses, generate database of pairs of parent chromosomes, with probability proportional to the fitness of both parents.

rand_model_f()
Generate a random model chromosome and evaluate its F score.

random_initial_models(*num_models=5*)
Generate random models from the space of valid candidates.

Parameters **num_models** (*int*) – number of candidates to generate

Returns list new_models the randomly generated model names

random_models_sorted_by_f_score(*num_models=14*)
Generate a set of random models and sort them by F score.

selection(***kwargs*)
Wrapper for user's selected selection method.

Whatever method is called must return

- prescribed_chromosomes
- chromosomes_for_crossover - pairs

truncate_to_top_half(*model_fitnesses*, ***kwargs*)
Retain only the top-performing half of models considered at this generation, for consideration as parents to offspring on the subsequent generation.

Parameters **model_fitnesses** (*dict*) – the fitness of each model in this generation according to the chosen objective function.

5.2.1 Genetic Exploration Strategy

A base class for genetic algorithm incorporated into the *Exploration Strategy*.

class qmla.exploration_strategies.genetic_algorithms.genetic_exploration_strategy.**Genetic**(

Exploration Strategy where the model search is mediated through a genetic algorithm. Genetic algorithm is implemented through `qmla.GeneticAlgorithmQMLA`. This forms the base class for genetic algorithm applications within QMLA.

Parameters

- **exploration_rules** (*str*) – name of exploration strategy used
- **genes** (*list*) – terms which are permitted in the model search, which become genes in the chromomsomes of the genetic algorithm
- **true_model** (*str*) – name of the target model.

__plot_gene_pool_progression()
Succinct representation of the progression of gene pool with respect to F score.

_plot_correlation_fitness_with_f_score(*save_to_file=None*)
Show how the fitness of models at each generation progress in terms of F score.

_plot_fitness_v_fscore()
Plot fitness against f score

_plot_fitness_v_fscore_by_generation()
Plot fitness vs f score throughout generations of the genetic algorithm.

_plot_fitness_v_generation(*save_to_file=None*)
Plot progression of fitness against generations of the genetic algorithm.

_plot_gene_pool()
Show the F scores of all models in all generations

_plot_model_ratings()
Plot ratings of models on all generations, as determined by the RatingSystem

_plot_selection_probabilities()
Plot pie charts of the selection probabilities of prospective parents at each generation. Models are signified by their F score.

analyse_generation(*model_points*, *model_names_ids*, ***kwargs*)
Following a complete generation of the genetic algorithm, perform all necessary processing to enable construction of next set of models.

Parameters

- **model_points** (*dict*) – the number of Bayes factor comparisons for which each candidate within the generation was deemed superior against a contemporary model
- **model_names_ids** (*dict*) – mapping between models' names and their IDs from the QMLA environment; this enables analaysing further data passed from QMLA within kwargs.

check_tree_completed(*spawn_step*, ***kwargs*)
Genetic algorithm specific version of `qmla.ExplorationStrategy.check_tree_completed()`.

check_tree_pruned(***kwargs*)
Genetic algorithm specific version of `qmla.ExplorationStrategy.check_tree_pruned()`.

exploration_strategy_finalise()
Genetic algorithm specific version of `qmla.ExplorationStrategy.exploration_strategy_finalise()`.

f_score_from_chromosome_string(*chromosome*)
F1 score between chromosome and true model

f_score_model_comparison(*test_model*, *target_model=None*, *beta=1*)
Get F score of candidate model, measure of overlap between the terms of the candidate and target model

Parameters

- **test_model** (*str*) – name of candidate model
- **target_model** (*str*) – name of target model, if None, assumed that target is self.true_model
- **beta** (*float*) – relative importance of precision to sensitivity. in general this is F-beta score, usually beta = 1

static gene_pool_progression(*gene_pool*, *ax*, *f_score_cmap=None*, *draw_cbar=True*, *cbar_ax=None*)
Method for plotting succinct summary of progression of gene pool with respect to F score.

generate_models(*model_list*, ***kwargs*)
Model generation using genetic algorithm.

Follows rules of *generate_models()*.

hamming_distance_model_comparison(*test_model*, *target_model=None*)
Compare test_model with target_model by Hamming distance

nominate_champions()
Choose model with highest fitness on final generation

plot_generational_metrics()
Show various metrics across all generations

set_specific_plots(***kwargs*)
Genetic algorithm specific version of `qmla.ExplorationStrategy.set_specific_plots()`.

CHAPTER

SIX

TUTORIAL

Here we provide a complete example of how to run the framework, including how to implement a custom *Exploration Strategy* (*ES*), and generate/interpret analysis.

6.1 Installation

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

1. these instructions are tested for Linux and presumed to work on Mac, but untested on Windows. It is likely some of the underlying software (redis servers) can not be installed on Windows, so running on *Windows Subsystem for Linux* is advised.
2. Python development tools are required by some packages: if the `pip install -r requirements` fail, here are some possible solutions.
3. Here we recommend using a virtual environment to manage the *QMLA* ecosystem; a resource for managing virtual environments is virtualenvwrapper. If using `virtualenvwrapper`, generate and activate a `venv` and disregard step 2 below.
4. In the following installation steps, ensure to replace `python3.6` with your preferred Python version. Python3.6 (or above) is preferred.

The steps of preparing the codebase are

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

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

# Ensure access to python dev tools
sudo apt-get install python3.6-dev

# make directory for QMLA
cd
mkdir qmla_test
```

(continues on next page)

(continued from previous page)

```
cd qmla_test

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

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

# Install dependencies
# Note some packages demand others are installed first,
# so are in a separate file.
cd QMLA
pip install -r requirements.txt
pip install -r requirements_further.txt
```

Note there may be a problem with some packages in the arising from the attempt to install them all through a single call to `pip install`. Ensure these are all installed before proceeding. When all of the requirements are installed, test that the framework runs. *QMLA* uses databases to store intermittent data: we must manually initialise the database. Run the following (note: here we list `redis-4.0.8`, but this must be corrected to reflect the version installed on the user's machine in the above setup section):

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

which should give something like Fig. 6.1.

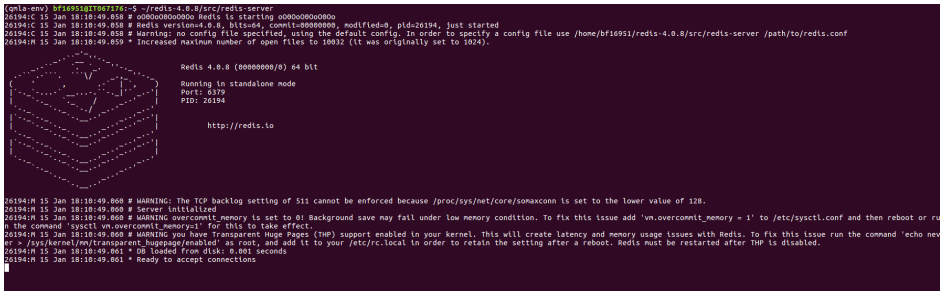

Fig. 6.1: Terminal running `redis-server`.

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

```
#!/bin/bash

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

6.2 Custom exploration strategy

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

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

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

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

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

# inside qmla/exploration_strategies, add to the existing
```

(continues on next page)

(continued from previous page)

```
# __init__.py
from qmla.exploration_strategies.custom_es import *
```

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

$$\begin{aligned} \boldsymbol{\alpha} &= (\alpha_{1,2} \quad \alpha_{2,3} \quad \alpha_{3,4}) \\ \boldsymbol{T} &= \left(\hat{\sigma}_z^1 \otimes \hat{\sigma}_z^2 \quad \hat{\sigma}_z^2 \otimes \hat{\sigma}_z^3 \quad \hat{\sigma}_z^3 \otimes \hat{\sigma}_z^4\right) \\ &\implies \hat{H}_0 = \hat{\sigma}_z^{(1,2)} \hat{\sigma}_z^{(2,3)} \hat{\sigma}_z^{(3,4)} \end{aligned} \tag{6.1}$$

QMLA interprets models as strings, where terms are separated by +, and parameters are implicit. So the target model in (6.1) will be given by

```
pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4
```

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

```
class ExampleBasic(
    exploration_strategy.ExplorationStrategy
):

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

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

    def generate_models(self, **kwargs):

        self.log_print(["Generating models; spawn step {}".format(self.spawn_step)])
        if self.spawn_step == 0:
            # chains up to 4 sites
            new_models = [
```

(continues on next page)

(continued from previous page)

```
                'pauliSet_1J2_zJz_d4',
                'pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4',
                'pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+pauliSet_3J4_zJz_d4',
            ]
            self.spawn_stage.append('Complete')

        return new_models
```

To run the example *ES* for a meaningful test, return to the `local_launch.sh` script above, but change some of the settings:

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

Run locally again then move to the results directory as in as in *Installation*. Note this will take up to 15 minutes to run. This can be reduced by lowering the values of `particles`, `experiments`, which is sufficient for testing but note that the outcomes will be less effective than those presented in the figures of this section.

6.3 Analysis

QMLA stores results and generates plots over the entire range of the algorithm, i.e. the run, instance and models. The depth of analysis performed automatically is set by the user control `plot_level` in `local_launch.sh`; for `plot_level=1` , only the most crucial figures are generated, while `plot_level=5` generates plots for every individual model considered. For model searches across large model spaces and/or considering many candidates, excessive plotting can cause considerable slow-down, so users should be careful to generate plots only to the degree they will be useful. Next we show some examples of the available plots.

6.3.1 Model analysis

We have just run *QHL* for the model in (6.1) for a single instance, using a reasonable number of particles and experiments, so we expect to have trained the model well. *Instance*-level results are stored (e.g. for the instance with `qmla_id=1`) in `Jan_01/01_23/instances/qmla_1`. Individual models' insights can be found in , e.g. the model's `leaning_summary` (Fig. 6.2), and in `dynamics` (Fig. 6.3).

6.3.2 Instance analysis

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

Reconfigure a subset of the settings in the `local_launch.sh` script and run it again:

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

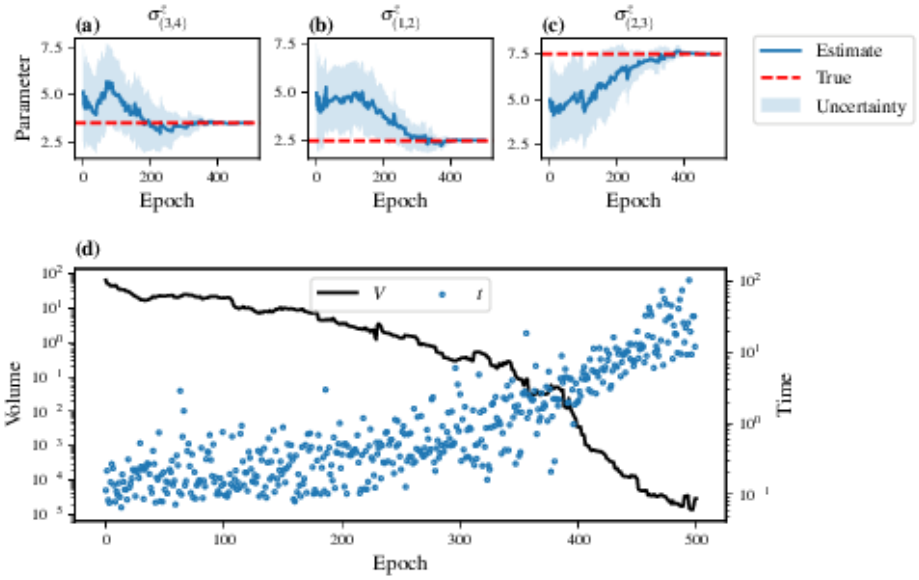

Fig. 6.2: The outcome of *QHL* for the given model. Subfigures (a)-(c) show the estimates of the parameters. (d) shows the total parameterisation volume against experiments trained upon, along with the evolution times used for those experiments.

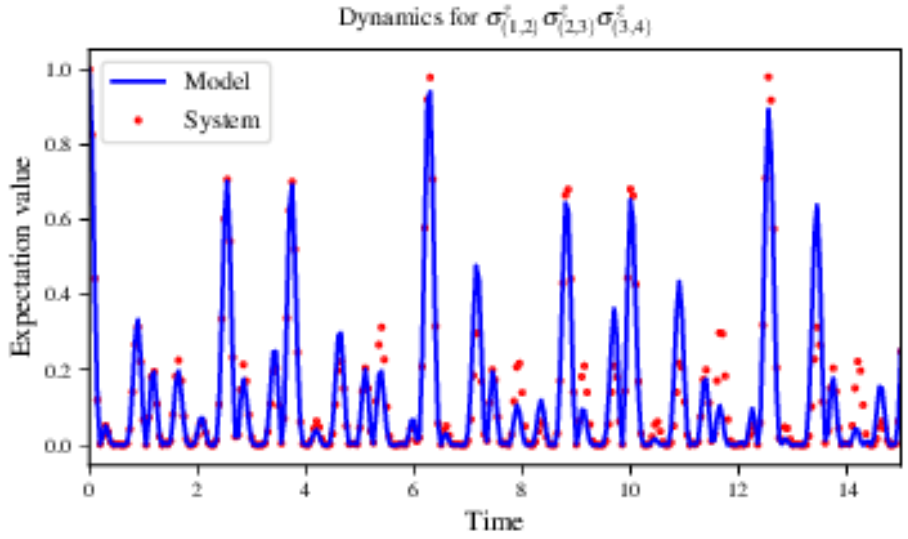

Fig. 6.3: The model's attempt at reproducing dynamics from $\hat{H}_0$.

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

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

Figures of interest here show the composition of the models (Fig. 6.4), as well as the *BF* between candidates (Fig. 6.5). Individual model comparisons – i.e. *BF* – are shown in Fig. 6.6, with the dynamics of all candidates shown in Fig. 6.7. The probes used during the training of all candidates are also plotted (Fig. 6.8).

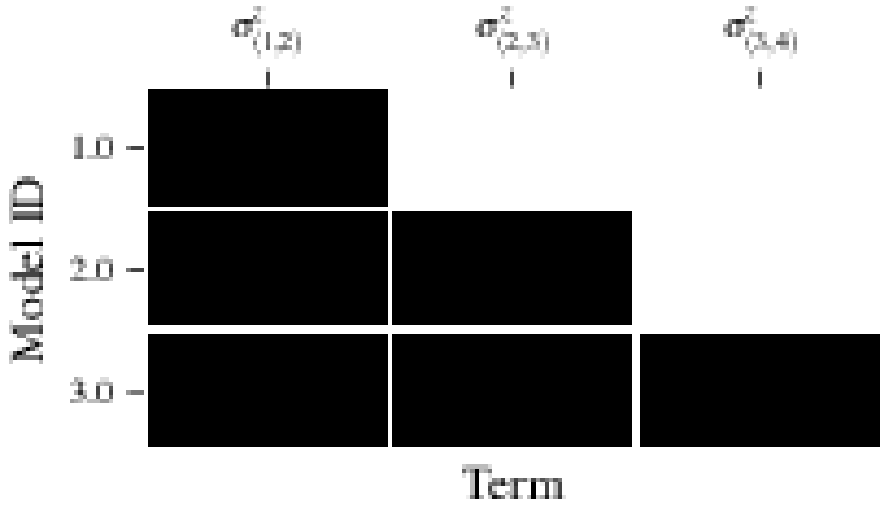

Fig. 6.4: `composition_of_models`: constituent terms of all considered models, indexed by their model IDs. Here model 3 is $\hat{H}_0$

6.3.3 Run analysis

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

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

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

Run level analysis are held in the main results directory and several sub-directories created by the script. For testing, here we recommend running a number of instances with very few resources so that the test finishes quickly (about ten minutes). The results will therefore be meaningless, but allow for elucidation of the resultant plots. First, reconfigure some settings of `local_launch.sh` and launch again.

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

Some of the generated analysis are shown in the following figures. The number of instances for which each model was deemed champion, i.e. their *win rates* are given in Fig. 6.9. The *top models*, i.e. those with highest win rates, analysed

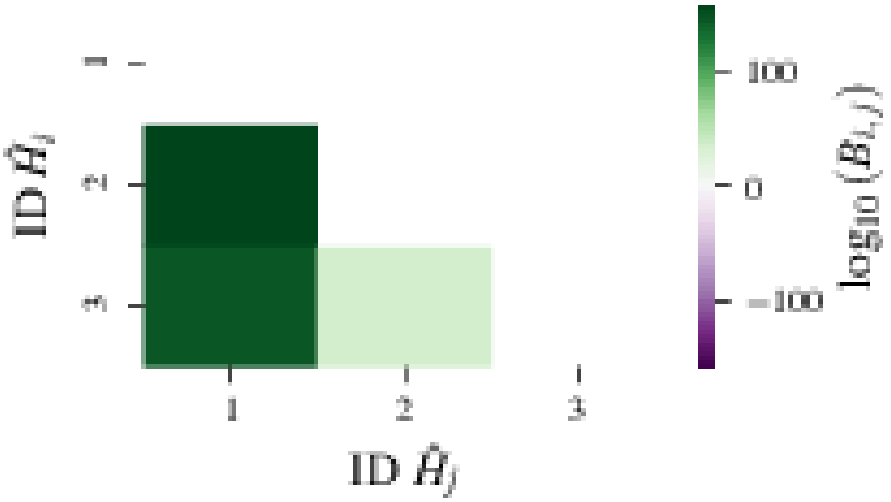

Fig. 6.5: `bayes_factors`: comparisons between all models are read as $B_{i,j}$ where i is the model ID on the y-axis and j on the x-axis. Thus $B_{ij} > 0$ (< 0) indicates $\hat{H}_i$ $(\hat{H}_j)$, i.e. the model on the y-axis (x-axis) is the stronger model.

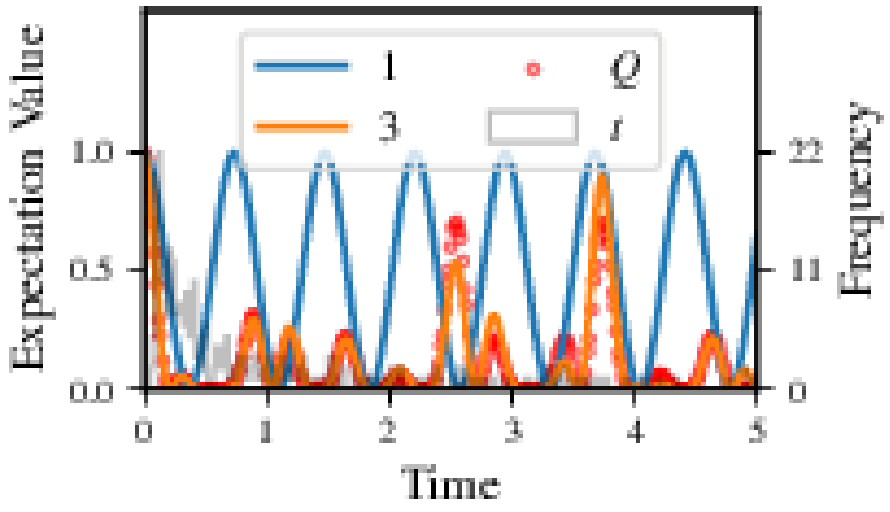

Fig. 6.6: `comparisons/BF_1_3`: direct comparison between models with IDs 1 and 3, showing their reproduction of the system dynamics (red dots, Q, as well as the times (experiments) against which the *BF* was calculated.

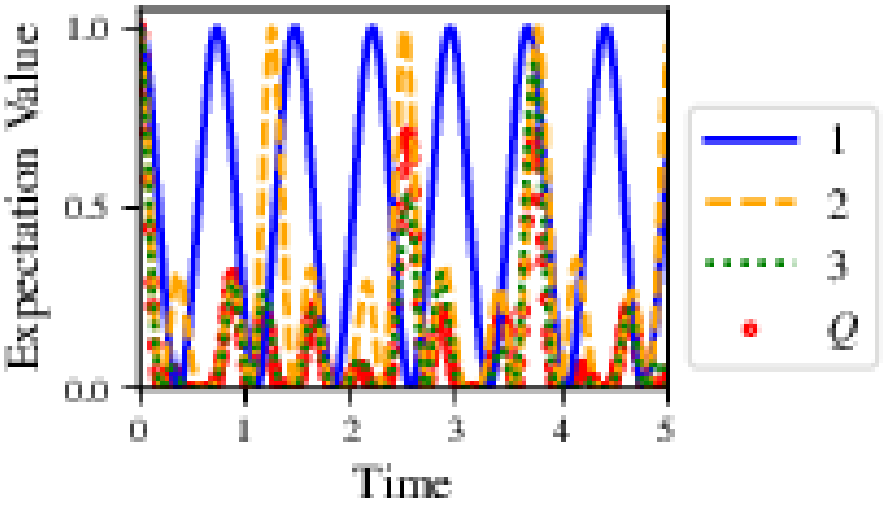

Fig. 6.7: `branches/dynamics_branch_1`: dynamics of all models considered on the branch compared with system dynamics (red dots, Q)

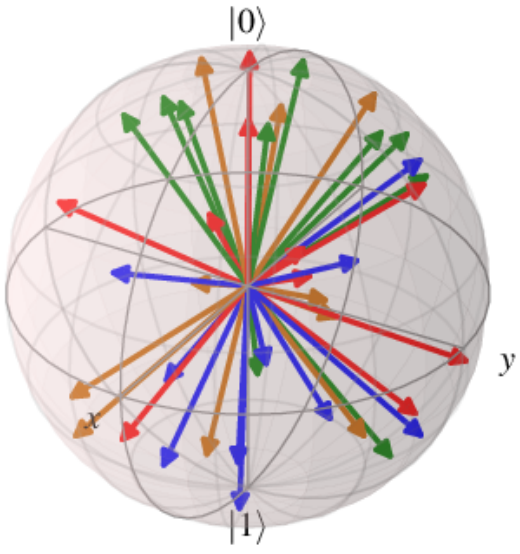

Fig. 6.8: `probes_bloch_sphere`: probes used for training models in this instance (only showing 1-qubit versions).

further: the average parameter estimation progression for $\hat{H}_0$ – including only the instances where $\hat{H}_0$ was deemed champion – are shown in Fig. 6.10. Irrespecitve of the champion models, the rate with which each term is found in the champion model ($\hat{t} \in \hat{H}'$) indicates the likelihood that the term is really present; these rates – along with the parameter values learned – are shown in Fig. 6.11. The champion model from each instance can attempt to reproduce system dynamics: we group together these reproductions for each model in Fig. 6.12.

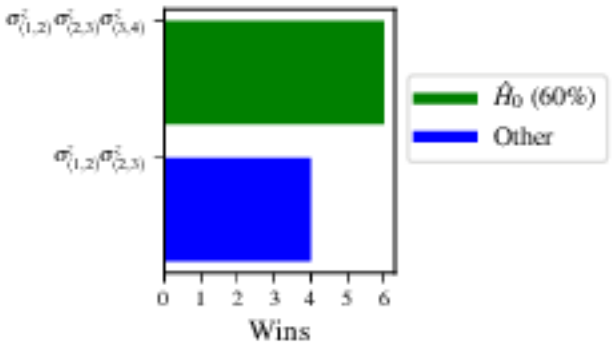

Fig. 6.9: `performace/model_wins`: number of instance wins achieved by each model.

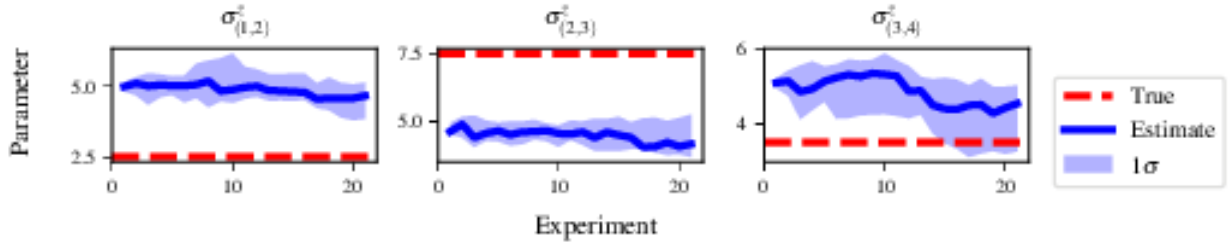

Fig. 6.10: `champion_models/params_params_pauliSet_1J2_zJz_d4+pauliSet_2J3_zJz_d4+pauliSet_3J4_z` parameter estimation progression for the true model, only for the instances where it was deemed champion.

6.4 Parallel implementation

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

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

```
#!/bin/bash

##### ------------------------------------------------- #####
# QMLA run configuration
##### ------------------------------------------------- #####
```

(continues on next page)

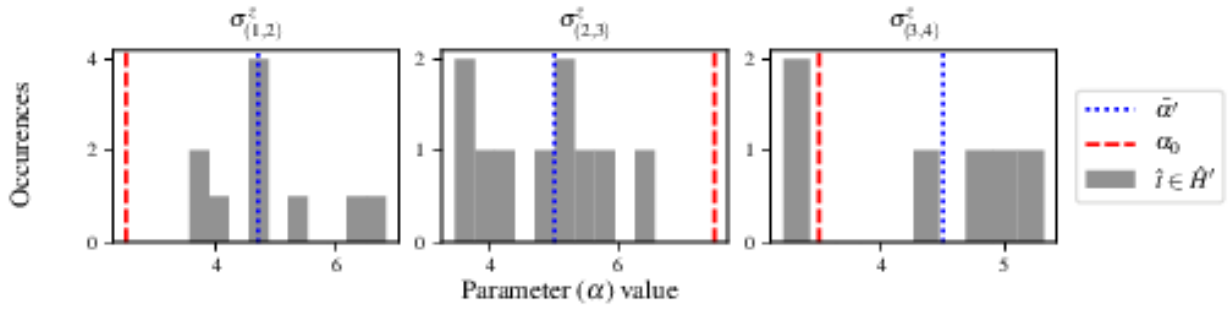

Fig. 6.11: `champion_models/terms_and_params`: histogram of parameter values found for each term which appears in any champion model, with the true parameter (α_0) in red and the median learned parameter ($\bar{\alpha}'$) in blue.

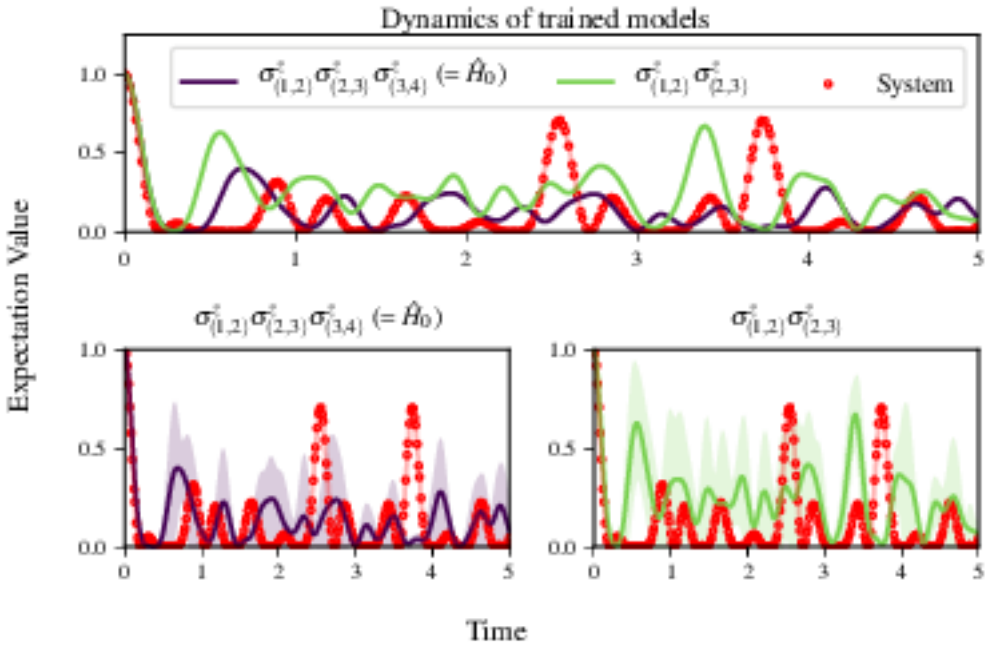

Fig. 6.12: `performance/dynamics`: median dynamics of the champion models. The models which won most instances are shown together in the top panel, and individually in the lower panels. The median dynamics from the models' learnings in its winning instances are shown, with the shaded region indicating the 66% confidence region.

(continued from previous page)

```
num_instances=10 # number of instances in run
run_qhl=0 # perform QHL on known (true) model
run_qhl_multi_model=0 # perform QHL for defined list of models
experiments=250
particles=1000
plot_level=5


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

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

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

```
module load tools/redis-4.0.8

...

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

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

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

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

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

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

6.5 Customising exploration strategies

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

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

6.5.1 Greedy search

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

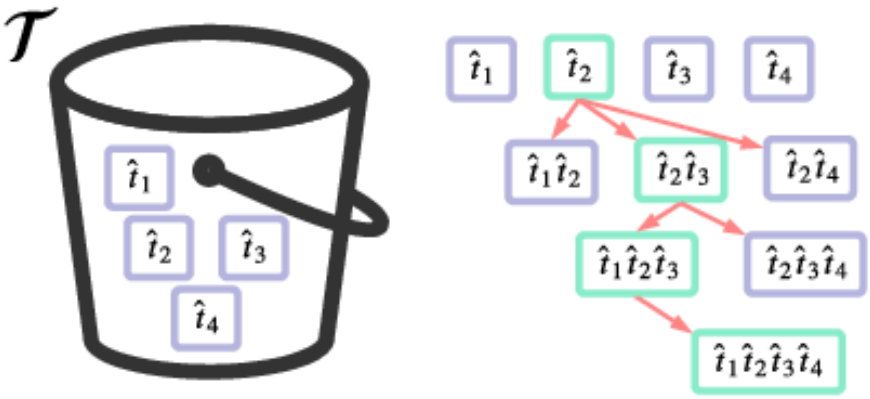

Fig. 6.13: Greedy search mechanism. **Left**, a set of primitive terms, $\mathcal{T}$, are defined in advance. **Right**, models are constructed from $\mathcal{T}$. On the first branch, the primitve terms alone constitute models. Thereafter, the strongest model (marked in green) from the previous branch is combined with all the unused terms.

We can compose an *ES* using these rules, say for

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

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

(continues on next page)

(continued from previous page)

```
    """

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

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

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

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

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

        return new_models

def greedy_add(
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

    return new_models
```

We advise reducing `plot_level` to 3 to avoid excessive/slow figure generation. This run can be implemented locally or in parallel as described above, and analysed through the usual `analyse.sh` script, generating figures in accordance with the `plot_level` set by the user in the launch script. Outputs can again be found in the instances subdirectory, including a map of the models generated (`fig:greedy_model_composition`), as well as the branches they reside on, and the Bayes factors between candidates, `fig:greedy_branches`.

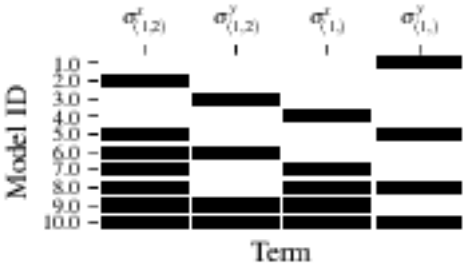

Fig. 6.14: `composition_of_models`

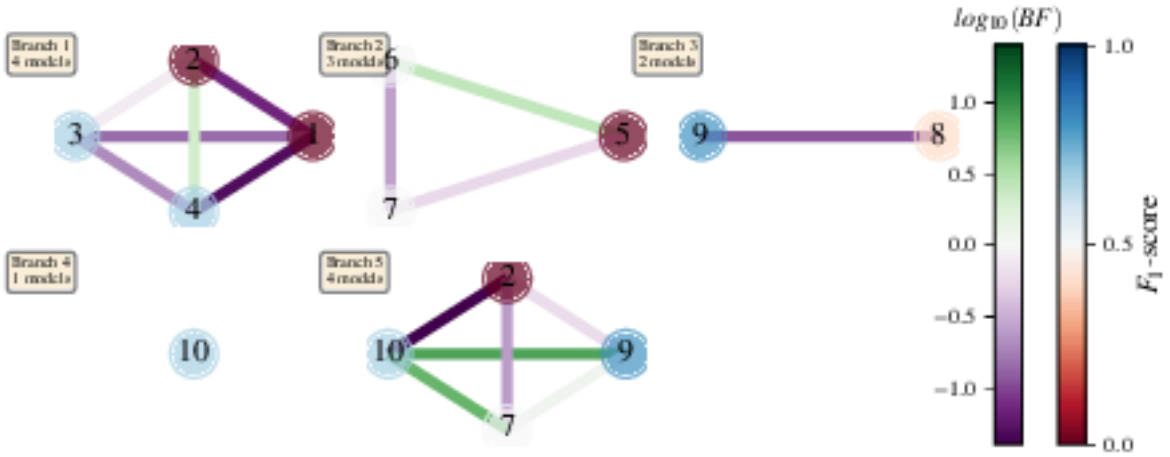

Fig. 6.15: `graphs_of_branches_ExampleGreedySearch`: shows which models reside on each branches of the exploration tree. Models are coloured by their F-score, and edges represent the *BF* between models. The first four branches are equivalent to those in Fig. 6.13, while the final branch considers the set of branch champions, in order to determine the overall champion.

6.5.2 Tiered greedy search

We provide one final example of a non-trivial *ES*: tiered greedy search. Similar to the idea of *Greedy search*, except terms are introduced hierarchically: sets of terms $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ are each examined greedily, where the overall strongest model of one tier forms the seed model for the subsequent tier. A corresponding :term:'Exploration Strategy' is given as follows.

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

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

    """

    def __init__(
        self,
        exploration_rules,
        **kwargs
    ):
        super().__init__(
            exploration_rules=exploration_rules,
            **kwargs
        )
        self.true_model = 'pauliSet_1_x_d3+pauliSet_1J2_yJy_d3+pauliSet_1J2J3_zJzJz_d3
↪'
        self.initial_models = None
        self.term_tiers = {
            1 : ['pauliSet_1_x_d3', 'pauliSet_1_y_d3', 'pauliSet_1_z_d3' ],
            2 : ['pauliSet_1J2_xJx_d3', 'pauliSet_1J2_yJy_d3', 'pauliSet_1J2_zJz_d3'],
```

(continues on next page)

(continued from previous page)

```
            3 : ['pauliSet_1J2J3_xJxJx_d3', 'pauliSet_1J2J3_yJyJy_d3', 'pauliSet_
↪1J2J3_zJzJz_d3'],
        }
        self.tier = 1
        self.max_tier = max(self.term_tiers)
        self.tier_branch_champs = {k : [] for k in self.term_tiers}
        self.tier_champs = {}
        self.prune_completed_initially = True
        self.check_champion_reducibility = True

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

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

        elif "getting_tier_champ" in self.spawn_stage[-1]:
            previous_branch_champ = model_list[0]
            self.log_print([
                "Tier champ for {} is {}".format(self.tier, model_list[0])
            ])
            self.tier_champs[self.tier] = model_list[0]
            self.tier += 1
            self.log_print(["Tier now = ", self.tier])
            self.spawn_stage.append(None) # normal processing

            if self.tier > self.max_tier:
                self.log_print(["Completed tree for ES"])
                self.spawn_stage.append('Complete')
                return list(self.tier_champs.values())
        else:
            self.log_print([
                "Spawn stage:", self.spawn_stage
            ])

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

        if len(new_models) == 0:
            # no models left to find - get champions of branches from this tier
```

(continues on next page)

(continued from previous page)

```
            new_models = self.tier_branch_champs[self.tier]
            self.log_print([
                "tier champions: {}".format(new_models)
            ])
            self.spawn_stage.append("getting_tier_champ_{}".format(self.tier))
        return new_models

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


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

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

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

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

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

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

    return new_models
```

with corresponding results in [fig:example_es_tiered_greedy].

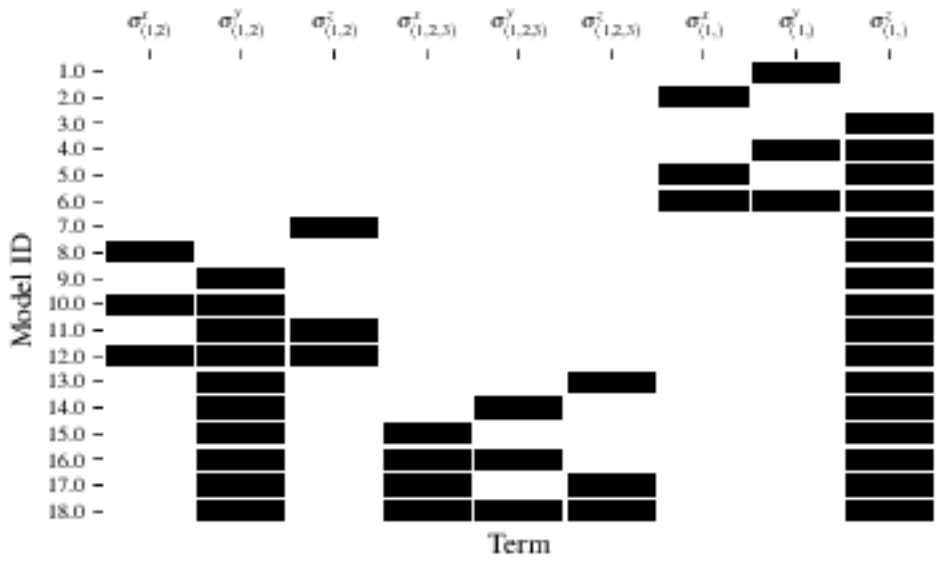

Fig. 6.16: `composition_of_models`

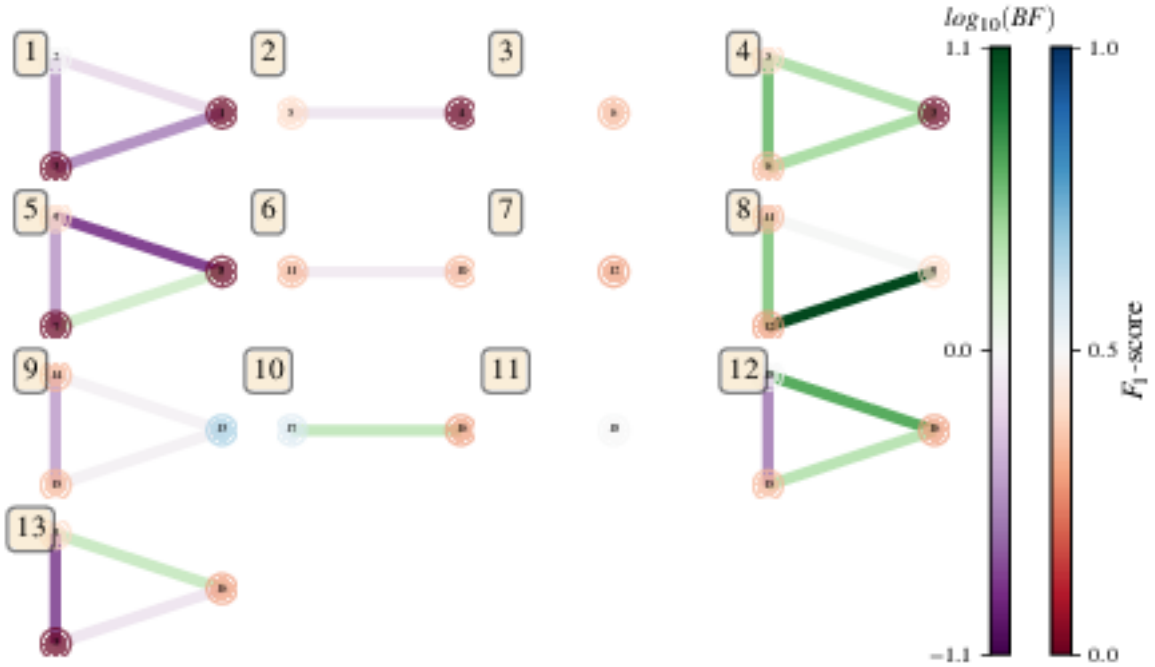

Fig. 6.17: `graphs_of_branches_ExampleGreedySearchTiered`: shows which models reside on each branches of the exploration tree. Models are coloured by their F-score, and edges represent the *BF* between models. In each tier, three branches greedily add terms, and a fourth branch considers the champions of the first three branches in order to nominate a tier champion. The final branch consists only of the tier champions, to nominate the global champion, $\hat{H}'$.

CHAPTER

SEVEN

BIBLOGRAPHY

BIBLIOGRAPHY

[WGFC13a] Wiebe N., Granade C. E., Ferrie C. & Cory D. G. Hamiltonian Learning and Certification Using Quantum Resources. arXiv:1309.0876

[WGFC13b] Wiebe N., Granade C. E., Ferrie C. & Cory D. G. Quantum Hamiltonian Learning Using Imperfect Quantum Resources. arXiv:1311.5269

[GFWC12] Granade C. E., Ferrie C., Wiebe N. & Cory D. G. Robust online Hamiltonian learning. New Journal of Physics **14** 103013 (2012). **:doi:`10.1088/1367-2630/14/10/103013`**. arXiv:1207.1655.

[QInfer] Granade, Christopher, et al. "QInfer: Statistical inference software for quantum applications." Quantum 1 (2017): 5. https://github.com/QInfer/python-qinfer

[GFK20] Gentile, Andreas A., Flynn, Brian, et al. "Learning models of quantum systems from experiments". arXiv:2002.06169

[SCG13] Smeltzer, Benjamin and Childress, Lilian and Gali, Adam. 13C hyperfine interactions in the nitrogen-vacancy centre in diamond. New Journal of Physics **13** 025021 (2013).

[QMLA] Flynn, Brian. Quantum Model Learning Agent. https://github.com/flynnbr11/QMLA

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

Q

R

S

T

U