The Quantum Computing Company™

D-Wave Solver Properties and Parameters Reference

USER MANUAL

2020-08-13

Overview

This document describes the solvers available in the D-Wave system, listing their properties and the parameters they accept with a problem submission.

CONTACT

Corporate Headquarters
3033 Beta Ave
Burnaby, BC V5G 4M9
Canada
Tel. 604-630-1428

US Office
2650 E Bayshore Rd
Palo Alto, CA 94303

Email: info@dwavesys.com

www.dwavesys.com

D-Wave User Manual 09-1169A-N

Notice and Disclaimer

Contents

1 About this Document

Solvers—compute resources to which you submit problems—accept certain parameters that control how the problem is run. This document lists the properties and parameters of:

- QPU (hardware) solvers
- QPU-like solvers, including the VFYC solver, optimizing emulators, and sampling emulators
- Leap's hybrid solvers
- Ising heuristic solvers

Note: Not all accounts have access to all solver types.

---

2 QPU (Hardware) Solvers

These solvers submit problems to and return results from the D-Wave QPU. This section lists the properties of these solvers, lists the parameters they accept with a problem submission, and describes the answer format.

For a quick reference listing all parameters and how they interact with each other, see the *QPU Solver Parameters Matrix* section.

2.1 Properties

This section describes the properties of QPU solvers, in alphabetical order.

anneal_offset_step

Quantization step size of anneal offset values in normalized units.

anneal_offset_step_phi0

Quantization step size in physical units (annealing flux-bias units): Φ_0.

anneal_offset_ranges

Array of ranges of valid anneal offset values, in normalized offset units, for each qubit. The negative values represent the largest number of normalized offset units by which a qubit's anneal path may be delayed. The positive values represent the largest number of normalized offset units by which a qubit's anneal path may be advanced.

Check these ranges before using the *anneal_offsets* parameter.

annealing_time_range

Range of time, in microseconds, possible for one anneal (read). The lower limit in this range is the fastest quench possible for this solver. When adjusting the anneal schedule, using either the *annealing_time* or *anneal_schedule* parameter, ensure that you do not exceed the limits in this range.

category

Type of solver; for example, `qpu`.

chip_id

Name of the solver.

couplers

Couplers in the working graph. A coupler contains two elements [*q1*, *q2*], where both *q1* and *q2* appear in the list of working qubits, in the range [0, *num_qubits* - 1] and in ascending order (i.e., *q1* < *q2*). These are the couplers that can be programmed with nonzero *J* values; for example, `[[0,4],[1,4],[2,4],...]`

default_annealing_time

Default time, in microseconds, for one anneal (read). You can change the annealing time for a given problem by using the *annealing_time* or *anneal_schedule* parameters, but do not exceed the upper limit given by the *annealing_time_range* property.

default_programming_thermalization

Default time, in microseconds, that the system waits after programming the QPU for it to return to base temperature. This value contributes to the total *qpu_programming_time*, which is returned by SAPI with the problem solutions.

You can change this value using the *programming_thermalization* parameter, but be aware that values lower than the default accelerate solving at the expense of solution quality.

default_readout_thermalization

Default time, in microseconds, that the system waits after each state is read from the QPU for it to cool back to base temperature. This value contributes to the *qpu_delay_time_per_sample* field, which is returned by SAPI with the problem solutions.

extended_j_range

Extended range of values possible for the coupling strengths (quadratic coefficients), *J*, for this solver. Strong negative couplings may be necessary for some embeddings; however, such chains may require additional calibration through the *flux_biases* parameter to compensate for biases introduced by strong negative couplings. See also *per_qubit_coupling_range*.

Note: Flux-bias offsets and extended *J* ranges cannot be used with autoscaling or spin-reversal transforms.

---

h_gain_schedule_range

Range of the time-dependent gain applied to qubit biases for this solver. When setting this gain, using the *h_gain_schedule* parameter, ensure that you do not exceed the limits in this range.

h_range

Range of values possible for the qubit biases (linear coefficients), *h*, for this solver.

The *auto_scale* parameter, which rescales *h* and *J* values in the problem to use as much of the range of *h* (*h_range*) and the range of *J* (*j_range*) as possible, enables you to submit problems with values outside these ranges and have the system automatically scale them to fit.

j_range

Range of values possible for the coupling strengths (quadratic coefficients), *J*, for this solver.

The *auto_scale* parameter, which rescales *h* and *J* values in the problem to use as much of the range of *h* (*h_range*) and the range of *J* (*j_range*) as possible, enables you to submit problems with values outside these ranges and have the system automatically scale them to fit.

See also *extended_j_range*.

max_anneal_schedule_points

Maximum number of points permitted in a PWL waveform submitted to change the default anneal schedule. Check this value before defining a new schedule with the *anneal_schedule* parameter.

For reverse annealing, the maximum number of points allowed is one more than the number given in the *max_anneal_schedule_points* property.

max_h_gain_schedule_points

Maximum number of points permitted in a PWL waveform submitted to set a time-dependent gain on linear coefficients (qubit biases, see the *h* parameter) in the Hamiltonian. Check this value before using the *h_gain_schedule* parameter.

num_reads_range

Range of values possible for the number of reads that you can request for a problem; for example, `[1,1000]`.

num_qubits

Total number of qubits, both working and nonworking, in the QPU; for example, `2048`.

parameters

List of the parameters supported for the solver.

per_qubit_coupling_range

Coupling range permitted per qubit for this solver. Check this property when using an extended J range to strongly couple qubits in a chain. Strong negative couplings may be necessary for some embeddings; however, chains may require additional calibration through the *flux_biases* parameter to compensate for biases introduced by strong negative couplings. See also *extended_j_range*.

Note: Flux-bias offsets and extended J ranges cannot be used with autoscaling or spin-reversal transforms.

---

problem_run_duration_range

Range of time, in microseconds, that a problem can run.

The upper limit of this range is calculated according to the following formula:

$$Duration = ((P_1 + P_2) * P_3) + P_4 \quad (1)$$

where P_1, P_2, P_3, and P_4 are the values specified for the annealing_time, readout_thermalization, num_reads (samples), and programming_thermalization parameters, respectively.

programming_thermalization_range

Range of time, in microseconds, possible for the system to wait after programming the QPU for it to cool back to base temperature This value contributes to the total *qpu_programming_time*, which is returned by SAPI with the problem solutions.

You can change this value using the *programming_thermalization* parameter, but be aware that values lower than the default accelerate solving at the expense of solution quality. The default value for a solver is given in the *default_programming_thermalization* property.

qubits

Indices of the working qubits in the working graph. For example, `[0,1,2,3,...]`

Note: In a D-Wave QPU, the set of qubits and couplers that are available for computation is known as the *working graph*. The yield of a working graph is typically less than the total number of qubits and couplers that are fabricated and physically present in the QPU.

---

quota_conversion_rate

Rate at which user or project quota is consumed for the solver. Time is deducted from your quota according to:

$$\frac{num\_seconds}{quota\_conversion\_rate}$$

If your quota_conversion_rate is `1`, for example, then the rate of quota consumption is straightforward: 1 second used on a solver deducts 1 second from your quota.

Different solver types may consume quota at different rates.

readout_thermalization_range

Range of time, in microseconds, possible for the system to wait after each state is read from the QPU for it to cool back to base temperature. This value contributes to the *qpu_delay_time_per_sample* field, which is returned by SAPI with the problem solutions.

supported_problem_types

Indicates what problem types are supported for the solver. QPU and QPU-like solvers support the following energy-minimization problem types:

- `qubo`—Quadratic unconstrained binary optimization (QUBO) problems; use 0/1-valued variables.
- `ising`—Ising model problems; use −1/1-valued variables.

tags

May hold attributes about a solver that you can use to have a client program choose one solver over another.

For example, the following attribute identifies a solver as lower-noise:

```
"tags": ["lower_noise"]
```

topology

Indicates the topology `type` (`chimera` or `pegasus`) and `shape` of the QPU graph. For example, the following topology is a C16 Chimera graph, meaning that the QPU has 16 x 16 blocks of Chimera unit cells, and each unit cell has K4,4 connectivity:

```
{"type": "chimera", "shape": [16, 16, 4]}
```

vfyc

Flag indicating whether this solver is a VFYC solver.

2.2 Parameters

This section describes the parameters accepted by QPU solvers, in alphabetical order. See *QPU Solver Parameters Matrix* for a summary.

anneal_offsets

Provides offsets to annealing paths, per qubit.

Provide an array of anneal offset values, in normalized offset units, for all qubits, working or not. Use 0 for no offset. Negative values produce a negative offset (qubits are annealed *after* the standard annealing trajectory); positive values produce a positive offset (qubits are annealed *before* the standard trajectory). Before using this parameter, query the solver properties to see whether the *anneal_offset_ranges* property exists and, if so, to obtain the permitted offset values per qubit.

When using the Ocean tools, supply the values as a list or a tuple.

anneal_schedule

Introduces variations to the global anneal schedule. For a reverse anneal, use the *anneal_schedule* parameter with the *initial_state* and *reinitialize_state* parameters.

An anneal schedule variation is defined by a series of pairs of floating-point numbers identifying points in the schedule at which to change slope. The first element in the pair is

time t in microseconds; the second, normalized persistent current s in the range [0,1]. The resulting schedule is the piecewise-linear curve that connects the provided points.

The following rules apply to the set of anneal schedule points provided:

- Time t must increase for all points in the schedule.
- For forward annealing, the first point must be $(0, 0)$.
- For reverse annealing, the anneal fraction s must start and end at $s = 1$.
- In the final point, anneal fraction s must equal 1 and time t must not exceed the maximum value in the *annealing_time_range* property.
- The number of points must be ≥ 2.
  The upper bound is system-dependent; check the *max_anneal_schedule_points* property. For reverse annealing, the maximum number of points allowed is one *more* than the number given by this property.
- Additional rules that govern maximum slope vary by product; check the QPU properties document for your system.

The *max_anneal_schedule_points* property shows the maximum number of points permitted in an anneal schedule, and *default_annealing_time* shows the annealing time range for the solver.

When using the Ocean tools, provide the schedule as a list or tuple of two-element lists or tuples; for example: `[[0.0,0.0],[10.0,0.3],[20.0,1.0]]`

Note: The *annealing_time* and *anneal_schedule* parameters are mutually exclusive.

Note: Spin-reversal transforms are incompatible with reverse annealing.

annealing_time

Sets the duration (in microseconds) of quantum annealing time, per read. This value populates the *qpu_anneal_time_per_sample* field returned in the *timing* structure.

Must be a positive integer falling within the range given by the *annealing_time_range* solver property.

Note: The *annealing_time* and *anneal_schedule* parameters are mutually exclusive.

answer_mode

Indicates how to return answers from the solver; one of:

- `histogram` (default)—Answers returned as a histogram sorted in order of increasing energies. Duplicate answers are merged and include a *num_occurrences* field.

- `raw`—Answers returned individually in the order they were read from the solver. Use this setting if the returned time sequences are an important part of the solution data.

See also the *Answer Format* section below.

auto_scale

Indicates whether h and J values are rescaled:

- `true`—h and J values in the problem are rescaled to use as much of the range of h (*h_range*) and the range of J (*j_range*) as possible. When enabled, h and J values need not lie within the solver's range of h and J, but must still be finite.
- `false`—h and J values in the problem are used as is. If the h and J values are outside the range of the solver, problem submission fails.

For problems that use the regular J range of a solver, this parameter is enabled by default. For problems that use the extended J range, it is disabled (and cannot be enabled while the extended range is in use). See also *j_range* and *extended_j_range*.

Note: Flux-bias offsets and extended J ranges cannot be used with autoscaling or spin-reversal transforms.

Auto-scaling works as follows. Each QPU has an allowed range of values for the biases and strengths of qubits and couplers. Unless you explicitly disable auto-scaling, the values defined in your problem are adjusted to fit the entire available range, by dividing them by a positive (non-zero) factor defined as:

$$\max\left\{\max\left(\frac{\max(h)}{\max(h\_range)},0\right),\max\left(\frac{\min(h)}{\min(h\_range)},0\right),\max\left(\frac{\max(J)}{\max(J\_range)},0\right),\max\left(\frac{\min(J)}{\min(J\_range)},0\right)\right\}$$

The example below uses Ocean software to check a QPU's range of h and J before submitting a two-variable Ising problem to a QPU. In this example, *DWaveSampler* implicitly sets *auto_scale* to *True*, so the h and J values are automatically rescaled by $\frac{-3.6}{-2} = 1.8$.

```
>>> from dwave.system import DWaveSampler, EmbeddingComposite
>>> sampler = EmbeddingComposite(DWaveSampler(solver={'qpu': True}))
...
>>> sampler.sampler.child.properties['j_range']
[-1.0, 1.0]
>>> sampler.sampler.child.properties['h_range']
[-2.0, 2.0]
...
>>> h = {'a': -3.6, 'b': 2.3}
>>> J = {('a', 'b'): 1.5}
>>> sampleset = sampler.sample_ising(h, J)
```

Problems specified in QUBO form are always converted to Ising for the submitted QMIs. When using auto-scaling, the converted problem's h and J values are rescaled as described above. Note that bias values in the converted form, which have a dependency on the number of quadratic interactions in the QUBO, can be larger than the maximum bias of the original form. For example, the four-variable QUBO below, which has a maximum bias

value of 2,

$$\begin{bmatrix} 2 & 2 & 1.5 & 2 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & -1.0 \end{bmatrix}$$

when converted to an Ising model, has a bias with a value greater than 2.0, $h_1 = 2.375$, as shown below:

```
>>> import dimod
>>> Q = {(1, 1): 2, (2, 2): 1.5, (3, 3): -0.5, (4, 4): -1.0,
...      (1, 2): 2, (1, 3): 1.5, (1, 4): 2}
>>> dimod.qubo_to_ising(Q)
({1: 2.375, 2: 1.25, 3: 0.125, 4: 0.0},
 {(1, 2): 0.5, (1, 3): 0.375, (1, 4): 0.5},
 2.375)
```

Ocean software's samplers often have a *chain strength* parameter: because the QPU's qubits are sparsely connected, problem variables might be represented by more than one physical qubit (a "chain" of qubits), strongly coupled so as to return the same value. Typically, chains are generated by minor-embedding tools such as Ocean's *minorminer*. Setting a value for chain strength determines the values set for the couplers used in forming these chains. When using auto-scaling, the J values of chain couplers are scaled together with the given or converted J values. Similarly, if you disable auto-scaling, any chain strength you specify must result in coupling values within the allowed range for the QPU.

beta

Provides a value for the Boltzmann distribution parameter. Used when sampling postprocessing is enabled on D-Wave 2000Q and earlier systems.

Note: As in statistical mechanics, β represents inverse temperature: $1/(k_B T)$, where T is the thermodynamic temperature in kelvin and k_B is Boltzmann's constant. In the D-Wave software, postprocessing refines the returned solutions to target a Boltzmann distribution characterized by β, which is represented by a floating point number without units. When choosing a value for β, be aware that lower values result in samples less constrained to the lowest energy states. For more information on β and how it is used in the sampling postprocessing algorithm, see *Postprocessing Methods on D-Wave Systems*.

---

chains

Defines which qubits represent the same logical variable. Used only when postprocessing is enabled on D-Wave 2000Q and earlier systems. Ensures that all qubits in the same chain have the same value within each sample.

Provide the indices of the qubits that are in a chain. Qubits in a chain must be connected, and no qubit may appear more than once.

When using the Ocean tools, provide the qubit indices as a list of lists. The inner lists are the indices of the qubits in the same chain; for example, `[4,12,20,28,36,44]`. An example with two parallel horizontal 4-qubit chains is `[[4,12,20,28],[5,13,21,29]]`.

flux_biases

List of flux-bias offset values with which to calibrate a chain. Often required when using the extended J range to create a strongly coupled chain for certain embeddings. See also the *extended_j_range* and *per_qubit_coupling_range* properties.

Provide an array of flux-bias offset values, in normalized offset units[1], for all qubits, working or not. Use 0 for no offset. Before using this parameter, query the solver properties to determine whether *extended_J_range* and *per_qubit_coupling_range* properties exist.

When using the Ocean tools, supply the values as a list or a tuple.

Note: Flux-bias offsets and extended J ranges cannot be used with autoscaling or spin-reversal transforms.

---

flux_drift_compensation

Boolean flag indicating whether the D-Wave system compensates for flux drift. The procedure it follows to do so is described in detail in Appendix A of *Postprocessing Methods on D-Wave Systems*.

- `flux_drift_compensation=true` (default)—Compensate for flux drift.
- `flux_drift_compensation=false`—Do not compensate for flux drift. If you disable this parameter, you may want to apply flux-bias offsets manually; see *flux_biases*.

h

For Ising problems, the h values are the linear coefficients (biases) applied to the qubits in a problem.

Because the quantum annealing process[2] minimizes the energy function of the Hamiltonian, and h is multiplied with the answer, returned states tend toward the opposite sign as that provided as a bias. For example, if you bias a qubit with an h value of -1, it is more likely to have a final state of $+1$ in the solution.

A problem definition comprises both h and J values. For example, the following is a 7-qubit problem in which q_1 and q_4 are biased with -1 and q_0, q_2, q_3, q_4, and q_6 are biased with $+1$. Qubits q_0 and q_6 have a ferromagnetic coupling between them with a strength of -1.

```
h = [1, -1, 1, 1, -1, 1, 1]
J = {(0, 6): -1}
```

[1] Flux-biases applied to the qubit body are in units of Φ_0.
[2] QPU-like solvers emulate this process.

To see the supported h range for a solver, check its *h_range* property. The *auto_scale* parameter enables you to submit problems with values outside *h_range* and *j_range* and have the system automatically scale them to fit.

Note: For QPU solvers, the programming cycle programs the solver to the specified h and J values for a given Ising problem (or Q values for a given QUBO problem). However, since QPU precision is limited, the h and J values (or Q values) realized on the solver may deviate slightly from the requested values. For more information, see *Technical Description of the D-Wave Quantum Processing Unit*.

Note: For QUBO problems, use **Q** instead of h and J; see Q for more information and instructions on converting between the formulations.

---

h_gain_schedule

Sets a time-dependent gain for linear coefficients (qubit biases, see the *h* parameter) in the Hamiltonian. This parameter enables you to specify the $g(t)$ function in,

$$\mathcal{H}_{ising} = -\frac{A(s)}{2}\left(\sum_i \hat{\sigma}_x^{(i)}\right) + \frac{B(s)}{2}\left(g(t)\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j}\hat{\sigma}_z^{(i)}\hat{\sigma}_z^{(j)}\right) \tag{2}$$

where $\hat{\sigma}_{x,z}^{(i)}$ are Pauli matrices operating on a qubit q_i and h_i and $J_{i,j}$ are the qubit biases and coupling strengths.

This time-dependent gain, $g(t)$, is specified, similarly to the *anneal_schedule* parameter, by a series of pairs of floating-point numbers identifying points in the schedule at which to change the gain applied to *h*. The first element in the pair is time, t in microseconds; the second, the unitless g in the range *h_gain_schedule_range*. The resulting time-dependent gain is the piecewise-linear curve that connects the provided points over the same range of times as the *anneal_schedule*.

The following rules apply to the set of gain points provided:

- Time t, in microseconds, must increase for all points in the schedule.
- The first point of time must be zero, $t = 0.0$.
- The last point of time must match the last time in the *anneal_schedule*.
- The number of points must be ≥ 2.
  The upper bound is system-dependent; check the *max_anneal_schedule_points* property.
- Additional rules that govern maximum slope (i.e., how quickly $g(t)$ can change) vary by product; check the QPU properties document for your system.
- The range of gain values permitted is a property of the solver; check the *h_gain_schedule_range* for your solver.

The *max_anneal_schedule_points* property shows the maximum number of points permitted in an anneal schedule, *max_h_gain_schedule_points* the maximum number of gain changes

allowed, and *h_gain_schedule_range* shows the minimum and maximum supported gain values for the solver.

When left unspecified, $g(t)$ defaults to 1, which can be explicitly coded as

```
h_gain_schedule=[[0,1],[t_final,1]]
```

where *t_final* is the requested annealing time.

When using the Ocean tools, provide the schedule as a list or tuple of two-element lists or tuples; for example: `[[0.0,0.0],[10.0,0.25],[20.0,0.4]]`

initial_state

When using the reverse annealing feature, you must supply the initial state to which the system is set. When you include this parameter with the *anneal_schedule* parameter, this indicates that the requested anneal schedule change is also a reverse anneal.

The *initial_state* parameter specifies the initial classical state of all qubits. Provide (*qubit*, *state*) pairs, where *qubit* is the qubit index, *i*, and *state* is:

- -1 or 1—Ising problems, active qubits
- 0 or 1—QUBO problems, active qubits
- 3—Unused or inactive qubits

When using the Ocean tools, provide the dense set of (*qubit*, *state*) pairs as a list or tuple.

You have the option to reassert this state for each anneal, although this impacts timing; see also *reinitialize_state*.

Note: Spin-reversal transforms are incompatible with reverse annealing.

J

For Ising problems, the J values are the quadratic coefficients applied to the couplers in a problem. This parameter defines the type and the strength of the coupling between a pair of qubits:

- **J < 0**: Ferromagnetic coupling; coupled qubits prefer to be in the same state, $(1, 1)$ or $(-1, -1)$.
- **J > 0**: Antiferromagnetic coupling; coupled qubits prefer to be in opposite states, $(-1, 1)$ or $(1, -1)$.
- **J = 0**: No coupling; qubit states do not affect each other.

The larger the absolute value of J, the stronger the coupling.

A problem definition comprises both h and J values. The following is a 7-qubit problem with a coupling strength of -1 between q_0 and q_6:

```
h = [1, -1, 1, 1, -1, 1, 1]
J = {(0, 6): -1}
```

To see the supported J range for a solver, check its *j_range* property. The *auto_scale* parameter enables you to submit problems with values outside *h_range* and *j_range* range and have the system automatically scale them to fit. Some solvers support an extended J range for stronger couplings; however, be aware that you cannot use the extended J range with autoscaling or spin-reversal transforms. See *extended_j_range* for more information.

Note: For QPU solvers, the programming cycle programs the solver to the specified h and J values for a given Ising problem (or Q values for a given QUBO problem). However, since QPU precision is limited, the h and J values (or Q values) realized on the solver may deviate slightly from the requested values. For more information, see *Technical Description of the D-Wave Quantum Processing Unit*.

Note: For QUBO problems, use **Q** instead of h and J; see *Q* for more information and instructions on converting between the formulations.

---

max_answers

Specifies the maximum number of answers returned from the solver:

- If *answer_mode* is `histogram`—Total number of distinct answers.
- If *answer_mode* is `raw`—Limits the returned values to the first *max_answers* of *num_reads* samples. In this mode, *max_answers* should never be more than *num_reads*.

Must be an integer > 0; default is *num_reads*.

num_reads

Indicates the number of states (output solutions) to read[3] from the solver. Must be a positive integer; default is `1`.

num_spin_reversal_transforms

Specifies the number of spin-reversal transforms[4] to perform.

Use this parameter to specify how many spin-reversal transforms to perform on the problem. Valid values range from 0 (do not transform the problem; the default value) to a value equal to but no larger than the *num-reads* specified. If you specify a nonzero value, the system divides the number of reads by the number of spin-reversal transforms to determine how many reads to take for each transform. For example, if the number of reads is 10 and the number of transforms is 2, then 5 reads use the first transform and 5 use the second.

Applying a spin-reversal transform can improve results by reducing the impact of analog errors that may exist on the QPU. This technique works as follows: Given an n-variable

[3] Terms synonymous to *reads* are *anneals* and *samples*.
[4] Also known as *gauge transformations*.

Ising problem, we can select a random $g \in \{\pm 1\}^n$ and transform the problem via $h_i \mapsto h_i g_i$ and $J_{ij} \mapsto J_{ij} g_i g_j$. A spin-reversal transform does not alter the mathematical nature of the Ising problem. Solutions s of the original problem and s' of the transformed problem are related by $s'_i = s_i g_i$ and have identical energies. However, the sample statistics can be affected by the spin-reversal transform because the QPU is a physical object with asymmetries.

Spin-reversal transforms work correctly with postprocessing and chains. Majority voting happens on the original problem state, not on the transformed state.

Be aware that each transform reprograms the QPU; therefore, using more than 1 transform will increase the amount of time required to solve the problem. For more information about timing, see *Solver Computation Time*.

Note: Flux-bias offsets and extended J ranges cannot be used with autoscaling or spin-reversal transforms. Furthermore, spin-reversal transforms are incompatible with reverse annealing.

---

postprocess

Postprocessing optimization and sampling algorithms provide local improvements with minimal overhead to solutions obtained from the quantum processing unit (QPU).

Ocean software provides postprocessing tools, and you can optionally run postprocessing online on D-Wave 2000Q and earlier systems.

Defines what type of postprocessing the system runs online on raw solutions:

- "" (empty string)—No postprocessing (default). Note that if this option is selected for the VFYC solver, sampling postprocessing runs.
- `sampling`—Runs a short Markov-chain Monte Carlo (MCMC) algorithm with single bit-flips starting from each sample. The target probability distribution is a Boltzmann distribution at inverse temperature β.
- `optimization`—Performs a local search on each sample, stopping at a local minimum.

Note: For problems that use online VFYC solvers, postprocessing always runs. By default, sampling postprocessing runs using a default β value of 10.

When postprocessing is enabled, qubits in the same chain, defined by the *chains* parameter, are first set to the same value by majority vote. Postprocessing is performed on the logical problem but qubit-level answers are returned. For more information about postprocessing, see *Postprocessing Methods on D-Wave Systems*.

Note: Ocean software provides postprocessing tools; use these for Advantage systems.

---

programming_thermalization

Gives the time (in microseconds) to wait after programming the QPU for it to cool back to base temperature (i.e., post-programming thermalization time). Lower values accelerate solving at the expense of solution quality. Must be an integer. This value contributes to the total *qpu_programming_time*, which is returned by SAPI with the problem solutions.

The default value for a solver is given in the *default_programming_thermalization* property; the range of possible values is given in the *programming_thermalization_range* property.

Q

A quadratic unconstrained binary operation (QUBO) problem is defined using an upper-triangular matrix, $\mathbf{Q}$, which is an $N \times N$ matrix of real coefficients, and $\mathbf{x}$, a vector of binary variables. The diagonal entries of $\mathbf{Q}$ are the linear coefficients that bias the qubits (analogous to h, in Ising problems). The nonzero off-diagonal terms are the quadratic coefficients that define the strength of the coupling between variables (analogous to J, in Ising problems).

Input may be full or sparse. Both upper- and lower-triangular values can be used; (i, j) and (j, i) entries are added together. If a Q value is assigned to a coupler not present, an exception is raised. Only entries indexed by working couplers may be nonzero.

When using the Ocean tools, this parameter is a dictionary of QUBO coefficients.

QUBO problems are converted to Ising format before they run on the solver. Ising format uses h (qubit bias) and J (coupling strength) to represent the problem; see also *h* and *J*.

Before sending a problem, check the *h_range* and *j_range* properties for the ranges permitted for the solver. Be aware that problems with values outside the supported range will, by default, be scaled down to fit within the supported range; see the *auto_scale* parameter for more information. Some solvers support an extended J range for stronger couplings; however, be aware that you cannot use the extended J range with autoscaling or spin-reversal transforms. See *extended_j_range* for more information.

Note: For QPU solvers, the programming cycle programs the solver to the specified h and J values for a given Ising problem (or Q values for a given QUBO problem). However, since QPU precision is limited, the h and J values (or Q values) realized on the solver may deviate slightly from the requested values. For more information, see *Technical Description of the D-Wave Quantum Processing Unit*.

Problem Conversion

Conversion between QUBO and Ising problem formulations is as follows:

$$s = q2 - 1. \tag{3}$$

To convert the coefficients from QUBO to Ising:

$$J_{i,j} = \frac{1}{4}\mathbf{Q}_{\mathrm{i,j}}, \tag{4}$$

$$h_i = \frac{1}{2}\mathbf{Q}_{i,i} + \frac{1}{4}\sum_{i<j}\mathbf{Q}_{i,j}, \tag{5}$$

or from Ising to QUBO:

$$\mathbf{Q}_{i,j} = 4J_{i,j} \tag{6}$$

$$\mathbf{Q}_{i,i} = 2h_i - \frac{1}{2}\sum_{i<j}\mathbf{Q}_{i,j}. \tag{7}$$

For example, this two-qubit problem in QUBO format,

```
Q = (0,5): -10,
```

is as follows in Ising format:

```
h = [-2.5, 0, 0, 0, 0, -2.5], J = {(0, 5): -2.5}.
```

The energies for the corresponding solutions have the constant difference of

$$\frac{1}{2}\sum_{i}\mathbf{Q}_{i,i} + \frac{1}{4}\sum_{i<j}\mathbf{Q}_{i,j}. \tag{8}$$

readout_thermalization

Gives the time (in microseconds) to wait after each state is read from the QPU for it to cool back to base temperature (i.e., post-readout thermalization time). This value contributes to the *qpu_delay_time_per_sample* field, which is returned with the problem solutions. Must be an integer.

reduce_intersample_correlation

Reduces sample-to-sample correlations caused by the spin-bath polarization effect[5] by adding a delay between reads.

Boolean flag indicating whether the system adds a delay.

- `reduce_intersample_correlation=true`—Adds delay.
- `reduce_intersample_correlation=false` (default)—Does not add delay.

Important: Enabling this parameter drastically increases problem run times. To avoid exceeding the maximum problem run time configured for your system, limit the number of reads when using this feature. For more information on timing, see *Solver Computation Time*.

---

[5] See the *Technical Description of the D-Wave Quantum Processing Unit* for more information on this effect.

reinitialize_state

When using the reverse annealing feature, you must supply the initial state to which the system is set; see the *initial_state* parameter. If multiple reads are requested in a single call to the Solver API, you have two options for the starting state of the system. These are controlled by the *reinitialize_state* Boolean parameter:

- `reinitialize_state=true` (default)—Reinitialize the initial state for every anneal-readout cycle. Each anneal begins from the state given in the *initial_state* parameter. The amount of time required to reinitialize varies by system; typical D-Wave 2000Q systems require between 100 and 600 microseconds microseconds for this operation.
- `reinitialize_state=false`—Initialize only at the beginning, before the first anneal cycle. Each anneal (after the first) is initialized from the final state of the qubits after the previous cycle. Be aware that even this parameter is disabled, reverse annealing adds a small amount of time ($\approx 10\ \mu s$).

See also *anneal_schedule*.

Relevant only to the C client. Pointer to the `sapi_ReverseAnneal` structure, which defines the initial state of the system at the start of a reverse anneal.

2.3 Answer Format

The format of answers returned by QPU solvers depends on the setting of the *answer_mode* parameter:

- If `raw`—The answer contains two fields, *solutions* and *energies*. The *solutions* field is a list of lists; the inner lists all have length *num_qubits* and entries from $-1, +1$ for Ising problems or $0, 1$ for QUBO problems. Values of 3 denote used or inactive qubits. The *energies* field contains the energy of each corresponding solution.
- If `histogram`—The answer still contains the *solutions* and *energies* fields, but in this case the solutions are unique and sorted in increasing-energy order. An additional field, *num_occurrences*, indicates how many times each solution appeared.

3 QPU-Like Solvers

This section describes the "QPU-like" solvers, including virtual full-yield chip (VFYC) solvers, optimizing emulators, and sampling emulators. These solvers have similar properties and accept similar parameters as the QPU solvers. Differences are described below.

Note: Not all accounts have access to these solvers.

3.1 VFYC Solvers

A VFYC solver emulates a fully connected QPU working graph based on an idealized abstraction of the system. Through this solver, variables corresponding to a Chimera-structured or Pegasus-structured graph that are not representable on a specific working graph are determined via hybrid use of the QPU and the integrated postprocessing system, which effectively fills in any missing qubits and couplers that may affect the QPU. For more information on the VFYC solver and how it is integrated with the postprocessing system, see *Postprocessing Methods on D-Wave Systems*.

Note: VFYC solvers for D-Wave 2000Q systems are available server-side for some accounts.

The properties of a VFYC solver and the parameters it accepts are the same as for the QPU solvers described above, with the following differences:

- For problems that use the VFYC solver, postprocessing always runs. If you do not choose a postprocessing option, sampling postprocessing runs.
- When sampling postprocessing runs, the default β value is 10.

3.2 Optimizing and Sampling Emulators

Software solvers are useful for prototyping optimization or sampling algorithms that make multiple calls to the hardware. Optimizing emulator solvers solve the same type of optimization problems as the QPU, but using a classical software algorithm. Sampling emulator solvers mimic the probabilistic nature of the QPU to draw samples that can be used to train a probabilistic model.

These solvers have names ending with the following strings:

- `-sw_optimize`
- `-sw_sample`

Optimizing Emulators

This class of solvers is entirely deterministic, so the semantics of some parameters are different. The number of solutions returned is always the lesser of *max_answers* and *num_reads* $\times$ *num_programming_cycles*. The solutions returned are the lowest-energy states, sorted in increasing-energy order.

These solvers accept the following parameters, described above:

- *answer_mode*
- *max_answers*
- *num_reads*

The acceptable range and the default value of each field are given in the table below.

Field	Range	Default value
answer_mode	`histogram` or `raw`	`histogram`
max_answers	> 0	*num_reads*
num_reads	> 0	1

When *answer_mode* is `histogram`, the *num_occurrences* field contains all ones, except possibly for the lowest-energy solution. That first entry is set so that the sum of all entries is *num_reads* $\times$ *num_programming_cycles*.

Note: The *num_programming_cycles* parameter is deprecated and will be removed in a future release. Plan code updates accordingly.

Sampling Emulators

This class of solvers mimic the probabilistic nature of the QPU, drawing samples from a Boltzmann distribution; that is, state s is sampled with probability proportional to:

$$\exp(-\beta E(s))$$

where β is some parameter and $E(s)$ is the energy of state s.

These solvers accept the following parameters, described above:

- *answer_mode*
- *beta*
- *max_answers*
- *num_reads*
- *random_seed*

In addition to this list, the sampling emulators accept an additional parameter: *random-seed*. This parameter provides a random number generator seed. If provided, the solver solves the same problem with the same parameters each read thereby producing the same results every time. If no value is provided, a time-based seed is selected, and solutions will vary.

The acceptable range and the default value of each field are given in the following table.

Field	Range	Default value
answer_mode	`histogram` or `raw`	`histogram`
beta	Any finite value	3.0
max_answers	> 0	*num_reads*
num_reads	> 0	1
random_seed	>= 0	Randomly set

4 Leap's Hybrid Solvers

Note: Not all accounts have access to this type of solver.

Leap's quantum-classical hybrid solvers are intended to solve arbitrary application problems formulated as binary quadratic models (BQM).

These solvers, which implement state-of-the-art classical algorithms together with intelligent allocation of the quantum processing unit (QPU) to parts of the problem where it benefits most, are designed to accommodate even very large problems. Leap's solvers can relieve you of the burden of any current and future development and optimization of hybrid algorithms that best solve your problem.

These solvers have the following characteristics:

- There is no fixed problem structure. In particular, these solvers do not have properties *num_qubits*, *qubits*, and *couplers*.
- Only one solution is returned and it is not guaranteed to be optimal.
- Solver properties and parameters are entirely disjoint from those of other solvers.

4.1 Generally Available Solvers

The generally available hybrid solvers depend on your Leap™ account. Check your Leap dashboard to see which hybrid solvers are available to you.

4.2 Properties

This section describes the properties of Leap's solvers, in alphabetical order.

category

Type of solver. Hybrid solvers support the following categories:

- `hybrid`—quantum-classical hybrid; typically one or more classical algorithms run on the problem while outsourcing to a quantum processing unit (QPU) parts of the problem where it benefits most.

maximum_number_of_variables

Maximum number of problem variables accepted by the solver.

minimum_time_limit

Minimum required run time, in seconds, the solver must be allowed to work on the given problem. Specifies the minimum time required for the number of problem variables, as a piecewise-linear curve defined by a set of floating-point pairs. The first element in each pair is the number of problem variables; the second is the minimum required time. The minimum time for any particular number of variables is a linear interpolation calculated on two pairs that represent the relevant range for the given number of variables. For example, if `minimum_time_limit` for a hybrid solver were `[[1,0.1],[100,10.0],[1000,20.0]]`, then the minimum time for a 50-variable problem would be 5 seconds, the linear interpolation of the first two pairs that represent problems with between 1 to 100 variables.

maximum_time_limit_hrs

Maximum allowed run time, in hours, that can be specified for the solver.

quota_conversion_rate

Ratio of time charged to Leap account quotas between QPU and hybrid solver usage. For example, for a value of 20, using 20 seconds of hybrid solver time is has an equivalent cost to using 1 second of QPU time.

supported_problem_types

Indicates what problem types are supported for the solver. Hybrid solvers support the following energy-minimization problem types:

- `bqm`—binary quadratic model (BQM) problems; use 0/1-valued variables and −1/1-valued variables.

4.3 Parameters

This section describes the parameters accepted by Leap' s hybrid solvers, in alphabetical order. See *Summary of Hybrid Solver Parameters* for a summary and for the default values.

time_limit

Specifies the maximum run time, in seconds, the solver is allowed to work on the given problem. Can be a float or integer in the range defined by *minimum_time_limit* for the problem's number of variables.

4.4 Summary of Hybrid Solver Parameters

Hybrid solver parameters and their default values are summarized in the table below.

Table 1: Hybrid Solver Parameters

Parameter	Range	Default Value
time_limit	See *minimum_time_limit*	Problem dependent

5 Ising Heuristic Solvers

Note: Not all accounts have access to this type of solver.

The Ising heuristic solver is intended to solve complex problem structures. It has some important differences from other solvers:

- There is no fixed problem structure. In particular, this solver does not have the properties *num_qubits*, *qubits*, and *couplers*.
- Only one solution is returned and it is not guaranteed to be optimal.
- Solver properties and parameters are entirely disjoint from those of other solvers.
- It cannot be used with the QSage solver.

Note that this heuristic solver can handle problems of arbitrary structure.

5.1 Algorithm

The core of the heuristic solver is a local search based on optimizing low-treewidth subgraphs. In pseudocode:

```
function local_search(problem, x)
  stuck = 0
  e = evaluate(problem, x)
  while (time limit not exceeded) and (stuck <= local_stuck_limit)
    select random low-treewidth subproblem
    (new_e, x) = solve(subproblem, x)
    if subproblem is the entire problem
      return (new_e, x, exact=true)
    if new_e < e
      stuck = 0
    else
      stuck = stuck + 1
    e = new_e
  return (e, x, exact=false)
```

The *local_stuck_limit* value is a user-supplied parameter. What constitutes a "low-treewidth subproblem" is determined by the parameter *max_local_complexity*. The time limit is provided by the parameter *time_limit_seconds*.

To escape local minima, multiple copies of the solution are made and bits are randomly flipped.

```
function ising_heuristic(problem)
  x = random solution vector
  (e, x, exact) = local_search(problem, x)
  if exact
    return (e, x)
  best_e = current_e = e; best_x = current_x = x
  iter = 0
  while (time limit not exceeded) and (iter < iteration_limit)
```

```
    iter = iter + 1
    new_e = current_e; new_x = current_x
    for i = 1 to num_perturbed_copies
      x = current_x
      flip each bit of x with probability p(i)
      (e, x, exact) = local_search(problem, x)
      if exact
        return (e, x)
      if e < new_e
        new_e = e; new_x = x
    current_e = new_e; current_x = new_x
    if current_e < best_e
      best_e = current_e; best_x = current_x
  return (best_e, best_x)
```

In the `ising_heuristic` function, *iteration_limit* and *num_perturbed_copies* are user-provided parameters. The (`i`-dependent) bit flip probability `p(i)` is determined by the parameters *min_bit_flip_prob* and *max_bit_flip_prob*.

5.2 Solver Name

`ising-heuristic`

5.3 Properties

None, except for the common property *supported_problem_types*.

5.4 Parameters

This section describes the parameters accepted by Ising heuristic solvers, in alphabetical order. See *Summary of Ising Heuristic Solver Parameters* for a summary and for the default values.

Note: Many of these parameters described here require a high-level understanding of the heuristic algorithm. Parameter settings can wildly affect solver performance and solution quality. It is difficult in general to know what good values are a priori; defaults are selected to favor quicker run times over aggressive searches. Therefore, experimentation with these values is recommended.

---

iteration_limit

Specifies the maximum number of solver iterations, not including the initial local search. Must be an integer >= 0; default = 10.

local_stuck_limit

Specifies the number of consecutive local search steps that do not improve solution quality to allow before determining a solution to be a local optimum. Larger values produce more thorough local searches but increase run time. Must be an integer > 0; default = 8.

max_bit_flip_prob

Bit flip probability range. The probability of flipping each bit is constant for each perturbed solution copy but varies across copies. The probabilities used are linearly interpolated between *min_bit_flip_prob* and *max_bit_flip_prob*. Larger values allow more exploration of the solution space and easier escapes from local minima but may also discard nearly-optimal solutions.

Must be a number [0.0 1.0] and min_bit_flip_prob <= max_bit_flip_prob, default min_bit_flip_prob = 1.0 / 32.0, default max_bit_flip_prob = 1.0 / 8.0.

max_local_complexity

Specifies the maximum complexity of subgraphs used during local search. The run time and memory requirements of each step in the local search are exponential in this parameter. Larger values allow larger subgraphs (which can improve solution quality) but require much more time and space. Subgraph "complexity" here means treewidth+1.

Must be an integer > 0; default = 9.

min_bit_flip_prob

Bit flip probability range. The probability of flipping each bit is constant for each perturbed solution copy but varies across copies. The probabilities used are linearly interpolated between *min_bit_flip_prob* and *max_bit_flip_prob*. Larger values allow more exploration of the solution space and easier escapes from local minima but may also discard nearly-optimal solutions.

Must be a number [0.0 1.0] and min_bit_flip_prob <= max_bit_flip_prob, default min_bit_flip_prob = 1.0 / 32.0, default max_bit_flip_prob = 1.0 / 8.0.

num_perturbed_copies

Number of perturbed solution copies created at each iteration. Run time is linear in this value.

Must be an integer > 0; default = 4.

num_variables

Provides a lower bound on the number of variables. This solver can accept problems of arbitrary structure and the size of the solution returned is determined by the maximum variable index in the problem. The size of the solution can be increased by setting this parameter.

Must be an integer >= 0, default = 0.

random_seed

Provides a random number generator seed. When a value is provided, solving the same problem with the same parameters will produce the same results every read. If no value is provided, a time-based seed is selected.

The use of a wall clock-based timeout may in fact cause different results with the same *random_seed* value. If the same problem is run under different CPU load conditions (or on computers with different performance), the amount of work completed may vary despite the fact that the algorithm is deterministic. If repeatability of results is important, rely on the *iteration_limit* parameter rather than the *time_limit_seconds* parameter to set the stopping criterion.

Must be an integer >= 0; default is a time-based seed.

time_limit_seconds

Specifies the maximum wall clock time in seconds. Actual run times will exceed this value slightly.

Must be a number >= 0.0; default = 5.0.

5.5 Summary of Ising Heuristic Solver Parameters

Ising heuristic solver parameters and their default values are summarized in the table below.

Table 2: Ising Heuristic Solver Parameters

Parameter	Range	Default Value
iteration_limit	>= 0	10
local_stuck_limit	> 0	8
max_bit_flip_prob	[min_bit_flip_prob 1.0]	1/8
max_local_complexity	> 0	9
min_bit_flip_prob	[0.0 1.0]	1/32
num_perturbed_copies	> 0	4
num_variables	>= 0	0
random_seed	>= 0	Time-based seed
time_limit_seconds	>= 0.0	5

6 Quick Reference Tables

This section provides two quick reference tables: *Properties and Parameters Summary by Solver Type* and *QPU Solver Parameters Matrix*.

6.1 Properties and Parameters Summary by Solver Type

The following table summarizes the properties and parameters of each solver described above. For release-specific details of QPU solver parameters, see *QPU Solver Parameters Matrix*.

Note: Not all accounts have access to all solver types.

Solver Type	Properties	Parameters
QPU and VFYC solvers	• *anneal_offset_step* • *anneal_offset_step_phi0* • *anneal_offset_ranges* • *annealing_time_range* • *category* • *chip_id* • *couplers* • *default_annealing_time* • *default_programming_thermalization* • *default_readout_thermalization* • *extended_j_range* • *h_gain_schedule_range* • *h_range* • *j_range* • *max_anneal_schedule_points* • *max_h_gain_schedule_points* • *num_reads_range* • *num_qubits* • *parameters* • *per_qubit_coupling_range* • *problem_run_duration_range* • *qubits* • *quota_conversion_rate* • *readout_thermalization_range* • *supported_problem_types* • *tags* • *topology* • *vfyc*	• *anneal_offsets* • *anneal_schedule* • *annealing_time* • *answer_mode* • *auto_scale* • *beta* • *chains* • *flux_biases* • *flux_drift_compensation* • *h_gain_schedule* • *initial_state* • *max_answers* • *num_reads* • *num_spin_reversal_transforms* • *postprocess* • *programming_thermalization* • *reduce_intersample_correlation* • *readout_thermalization* • *reinitialize_state*
Optimizing emulators	• *couplers* • *num_qubits* • *num_reads_range* • *parameters* • *qubits* • *supported_problem_types*	• *answer_mode* • *max_answers* • *num_reads*
Sampling emulators	• *beta_range* • *couplers* • *default_beta* • *num_qubits* • *num_reads_range* • *parameters* • *qubits* • *supported_problem_types*	• *answer_mode* • *beta* • *max_answers* • *num_reads* • *random_seed*

Solver Type	Properties	Parameters
Hybrid solvers	• *time_limit*	• *category* • *maximum_number_of_variables* • *maximum_time_limit_hrs* • *minimum_time_limit* • *quota_conversion_rate* • *supported_problem_types*

Solver Type	Properties	Parameters
Ising heuristic solvers	• *supported_problem_types*	• *iteration_limit* • *local_stuck_limit* • *max_bit_flip_prob* • *max_local_complexity* • *min_bit_flip_prob* • *num_perturbed_copies* • *num_variables* • *random_seed* • *time_limit_seconds*

6.2 QPU Solver Parameters Matrix

The following table summarizes the QPU solver parameters, identifies on what platform and in what release they are available, provides guidelines on how parameters interact with each other and on timing,[1] and gives their default settings.

Notice that some parameters are not supported on D-Wave 2X and earlier platforms. Always check solver properties when using a new parameter.

[1] For more information on timing, see *Solver Computation Time*.

Table 3: QPU Solver Parameters Matrix

Parameter	Pltfrm./ Rel.	Parameter Compatibility	Timing Impact?	Default
anneal_offsets	2000Q, 2.5+		No	No offsets.
anneal_schedule	2000Q, 2.11+	Incompatible with *annealing_time*. Use with *initial_state* and *reinitialize_state* for reverse annealing.	Yes	No variation to the standard anneal schedule.
annealing_time	All	Incompatible with *anneal_schedule*.	Yes	Check properties.
answer_mode	All		No	`histogram`
auto_scale	All	Incompatible with *flux_biases* and extended *J* range.	Yes	`true`
beta	All		No	Check properties. For the VFYC solver, this is `10`.
chains	All	Use with *postprocess*.	No	No chains.
flux_biases	2000Q, 3.0+	Incompatible with *auto_scale* and *num_spin_reversal_transforms*.	No	No offsets.
flux_drift_compensation	2000Q, 3.0+		No	`true`
h	All		No	No bias.
h_gain_schedule	2000Q	Use with *anneal_schedule*.	No	`1`
initial_state	2000Q, 3.0+	Use with *anneal_schedule* for reverse annealing. Incompatible with *annealing_time*.	Yes	No initial state.
J	All		No	No coupling.
max_answers	All		No	*num_reads*
num_reads	All	The web server supports up to 1000 reads. Ocean client support up to 10,000.	Yes	`1`
num_spin_reversal_transforms	All	Incompatible with reverse annealing, *flux_biases*, and extended *J* range.	Yes	`0`
postprocess	All	Use with *chains*.	Yes	None.[2]
programming_thermalization	All		Yes	Check properties.
Q	All		No	N/A
qaot	All		Yes	Introduced by the system. Roughly 1 ms.
reduce_intersample_correlation	2000Q, 3.0+		Yes	`false`
readout_thermalization	All		Yes	Check properties.
reinitialize_state	2000Q, 3.0+	Use with *initial_state* and *reinitialize_state* for reverse annealing.	Yes	`true`

[2] The VFYC solver always runs postprocessing; its default is `sampling`.