SoftwareX 36 (2026) 103050

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

autotn: Automata-inspired construction of tensor-network operators from symbolic local rules

Aitor Morais [a,c,*], Izaskun Oregi [b], Iker Pastor [a], Eneko Osaba [c]

[a] University of Deusto (D4K), Bilbao, Spain
[b] Department of Applied Mathematics, University of the Basque Country (UPV/EHU), Vitoria-Gasteiz, Spain
[c] Tecnalia, Basque Research and Technology Alliance (BRTA), Derio, Spain

ARTICLE INFO

Keywords:
Tensor networks
Matrix product operators
Python software
Quantum many-body systems

ABSTRACT

Matrix product operators (MPOs) are a standard representation for structured operators in tensor-network simulations and quantum computing applications. However, converting symbolic descriptions of Hamiltonians and cost functions into MPO representations often requires substantial manual implementation, making the process error-prone and difficult to reproduce. This paper presents autotn, an open-source Python software tool that converts symbolic operator rules into MPO representations. Users define local operators and interaction rules in a high-level format, and the software generates the corresponding MPO along with validation utilities. In doing so, the software follows an automata-inspired approach that identifies repeated patterns in the symbolic description of the operator and reuses them to build a compact tensor-network representation, avoiding the need to construct each tensor by hand. The current implementation of autotn supports both diagonal operators, commonly encountered in combinatorial optimization problems, and general local matrix operators used in quantum many-body physics. The software is demonstrated on Max-Cut cost operators, long-range $XX + YY + Z$ spin Hamiltonians, and quantum clock models. By providing a reproducible workflow from symbolic model specification to validated MPO construction, autotn reduces the engineering effort required to develop, verify, and compare MPO constructions.

Required metadata

Current code version

Code metadata.

Nr.	Code metadata description	Current information
C1	Current code version	1.0.0
C2	Permanent GitHub link to code/repository used for this code version	GitHub tag v1.0.0.
C3	Legal Code License	MIT License.
C4	Code versioning system used	Git, with source code hosted on GitHub at aitormorais/tngenerator.
C5	Software code languages, tools, and services used	Python; NumPy; pytest for tests; Matplotlib for optional benchmark plots; setuptools for packaging.
C6	Compilation requirements, operating environments & dependencies	Python ≥ 3.10; NumPy. Test dependency: pytest. Optional benchmark plotting dependency: Matplotlib.
C7	If available Link to developer documentation/manual	Developer documentation is provided in the repository README.md.
C8	Support email for questions	aitor.morais@deusto.es.

* Corresponding author at: University of Deusto (D4K), Bilbao, Spain.
Email address: aitor.morais@deusto.es (A. Morais).

https://doi.org/10.1016/j.softx.2026.103050
Received 24 July 2026; Received in revised form 4 September 2026; Accepted 15 September 2026
Available online 23 September 2026
2352-7110/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Current executable software version

Executable software metadata.

Executable metadata	Value
Software name	`autotn`
Version	1.0.0
Programming language	Python
Main dependency	NumPy
Test command	`python -m pytest`
Repository	GitHub tag v1.0.0.

1. Motivation and significance

Many applications in combinatorial optimization, quantum computing, and quantum many-body physics are defined through sums of local interactions. In the weighted Max-Cut problem, for example, each edge contributes a term involving the variables associated with its two incident vertices. Similarly, quantum many-body Hamiltonians commonly combine local fields and interactions acting on one, two, or several sites [1,2]. In these domains, operators are typically specified using symbolic expressions composed of local operator terms and associated coefficients. Such representations are concise, intuitive, and closely aligned with the mathematical formulation of the underlying model.

A generic operator can be written as

$$\hat{H} = \sum_{\alpha=1}^{M} c_\alpha \bigotimes_{i=1}^{N} \hat{O}_i^{(\alpha)}, \tag{1}$$

where N denotes the number of sites, M is the number of interaction terms, and c_α is the coefficient associated with term α. For each site i, the local object $\hat{O}_i^{(\alpha)}$ represents either a local operator acting on site i or the identity when that site does not participate in the interaction. This notation is compact and closely reflects the mathematical structure of the underlying model, making it convenient for defining and interpreting both optimization objectives and physical Hamiltonians [3,4].

However, symbolic expressions alone are not sufficient for numerical calculations. Most computational frameworks require an explicit numerical representation of the operator. Constructing the corresponding dense matrix quickly becomes impractical because its size grows exponentially with the number of sites. Consequently, even operators defined by a relatively small set of symbolic terms can become prohibitively expensive to store and manipulate in their full matrix form. This motivates the use of numerical representations that preserve the local structure of the model without constructing the full operator [5].

To address this limitation, tensor-network (TN) representations provide compressed numerical descriptions that exploit the locality and repeated structure present in many operators. Among them, matrix product operators (MPOs, [6]) have become a standard representation for Hamiltonians, optimization cost functions, time-evolution operators, and other structured operators [1,2,6,7]. By decomposing a large operator into a sequence of local tensors connected by auxiliary bonds, MPOs can represent many physically relevant operators using substantially fewer parameters than a dense matrix representation.

Despite their widespread use, operators are seldom expressed directly as MPOs. Instead, researchers commonly begin with symbolic descriptions such as Eq. (1) and subsequently translate them into numerical MPO representations. This issue has motivated several approaches for systematic MPO construction. Finite-state-machine and automata-based descriptions provide a way to encode families of operator strings into compact MPO structures [3]. Other works construct generic MPOs through MPO arithmetic followed by compression, showing that efficient representations can be obtained for broad classes of operators [4]. Additional algorithmic schemes exploit graph-based or symmetry-aware constructions to generate MPOs automatically from structured symbolic input [8,9]. These developments highlight that MPO construction is not merely an implementation detail; it is the step that connects a mathematical model with the numerical representation required for computations.

The need for MPO construction arises within a mature ecosystem of TN and quantum-simulation software. ITensor provides a comprehensive TN library with mature MPO functionality [10]. TeNPy supports Python simulations of strongly correlated quantum systems [11]. Quimb combines TN algorithms with quantum-information and many-body utilities [12], while OpenFermion focuses on symbolic fermionic and qubit-operator workflows for quantum simulation [13]. Although existing TN libraries provide robust simulation capabilities, the step that converts symbolic model definitions into validated MPO representations is often left to user-specific implementations. This creates an engineering gap between the mathematical specification of an operator and its reliable numerical realization. In this work we present `autotn`, a software package designed to bridge this gap by providing a reproducible and automated MPO-construction workflow.

Within this ecosystem, ITensor's AutoMPO/OpSum interface is an established route for MPO construction: users specify sums of operator products, which are then converted into MPO objects inside the ITensor simulation framework [10]. `autotn` is not intended to replace this functionality. Its scope is different: it provides a lightweight Python workflow for translating symbolic local-operator rules into TN arrays independently of a particular tensor-network solver. In addition, `autotn` emphasizes construction diagnostics, automatic split-point selection, explicit tensor-shape metadata, and dense-reference validation on tractable benchmark instances. These features are intended to support reproducible prototyping and inspection of MPO encodings before they are used in larger simulation or optimization workflows. The exact MPO representations are derived from symbolic sums of local operator strings. The package provides a reproducible pipeline that transforms compact symbolic model specifications into MPOs, reducing the need for manual tensor assembly. Its design is inspired by the finite-automata approach of Crosswhite and Bacon [3], where automata-like state caching can be used to organize matrix product algorithms. The current implementation does not expose a formal finite-automaton language. Instead, it uses an automata-inspired construction in which groups of terms with common prefixes and suffixes define the virtual states of the TN.

Rather than replacing existing TN and symbolic-operator frameworks, `autotn` complements them by focusing on the automated generation and validation of MPO representations. Its intended use is reproducibility-oriented prototyping: users define local operators and symbolic interaction rules, while the software constructs the corresponding tensor network, selects an internal split point, and records structural metadata. The resulting MPOs can then be inspected, validated against dense references for computationally tractable instances, or integrated into larger simulation workflows.

The resulting workflow supports the development and evaluation of MPO encodings by integrating construction, structural diagnostics, and dense-reference validation within a common framework. Tasks that often require problem-specific implementations – such as split-

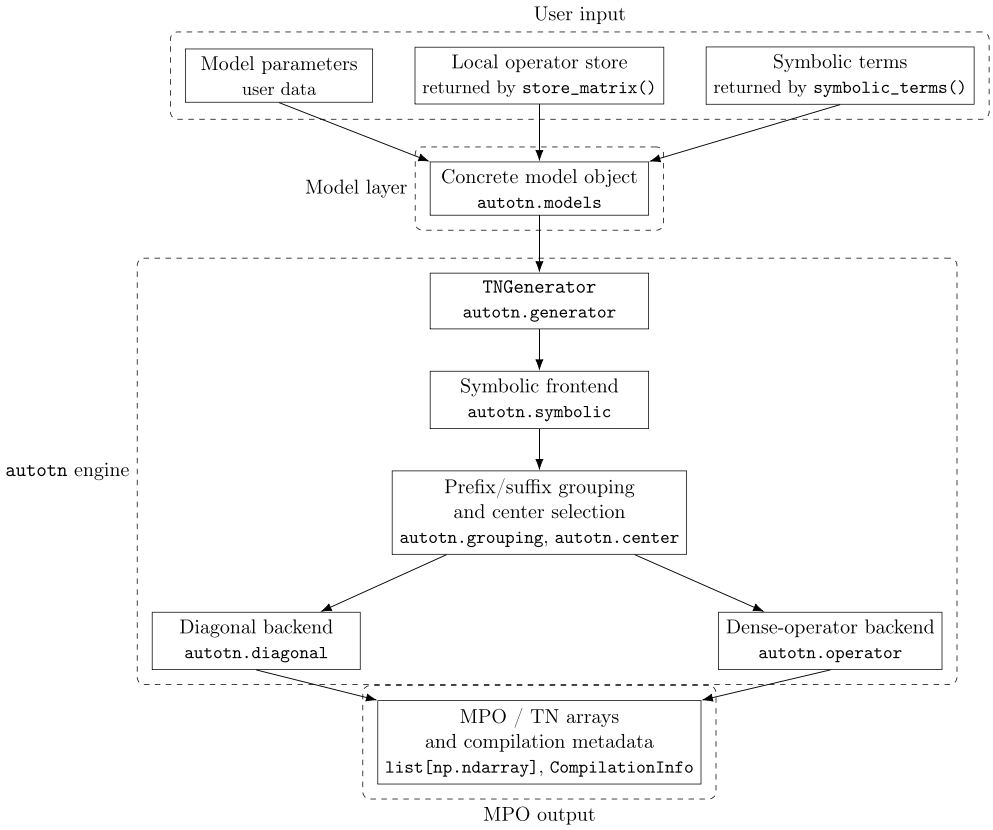


Fig. 1. Software architecture used by `autotn`. The user provides problem parameters and the symbolic operators. The model layer gathers input information for the rest of the package. The `autotn` engine processes this symbolic description, selects a construction route, and produces an MPO representation together with compilation metadata.

point selection, tensor-shape verification, and operator validation – can therefore be performed using the same software infrastructure. As a result, MPO representations can be developed, inspected, validated, and compared more systematically before being incorporated into larger simulation or optimization workflows.

2. Software description

This section describes how `autotn` transforms symbolic operators into MPO representations. We first describe the main software components and then follow the construction flow from model-specific input data to symbolic operator strings, center selection, backend compilation, and validation metadata. The goal is to clarify how the different modules interact, highlight the main design choices of the package, and focus on the functions that are most relevant to the construction process.

2.1. Workflow

The architecture of `autotn` is organized into several stages transforming user input into an MPO. As shown in Fig. 1, the workflow starts from the problem-specific model parameters provided by the user. These parameters are encapsulated in a model object implemented in `autotn.models`. The model object provides `store_matrix()` which contains operators in matrix or vector form, and `symbolic_terms()` which contains all the symbolic operator terms.

For model-based construction, users call the `TNGenerator` class. Given a model object, `TNGenerator` retrieves the local operator store, passes the model's `symbolic_terms()` method as the symbolic term generator, selects any model-provided default center strategy when appropriate, and delegates the remaining compilation workflow to `autotn.symbolic`. The symbolic frontend normalizes the symbolic description, infers the number of sites when it is not provided explicitly, and determines whether the local store contains diagonal vectors or dense local-operator matrices.

The automata-inspired part of the workflow appears in the prefix/suffix grouping and center-selection stage. Symbolic operator strings are grouped according to shared prefixes and suffixes, and these shared partial strings define the virtual states used to build the TN representation. The same prefix/suffix structure is also used to compare candidate split points before constructing the final tensors.

The construction is then dispatched to one of two backend modules. Diagonal local-vector descriptions are compiled by `autotn.diagonal`, while dense local-operator descriptions are compiled by `autotn.operator`. In both cases, the selected backend produces the final MPO/TN arrays together with compilation metadata describing the selected construction.

2.2. Software functionalities

`autotn` is organized around a compact set of construction, inspection, and validation tasks for TN operators. The current repository focuses on converting symbolic local-operator rules into concrete NumPy tensor arrays, while exposing enough diagnostics to compare models, split choices, and dense-reference checks on computationally tractable instances. The repository currently provides:

- construction of diagonal TN from symbolic local-vector rules;
- construction of dense local-operator TN from symbolic local-matrix rules;

- automatic split-point selection using exact shape estimates;
- validation utilities for computationally tractable instances by explicit contraction;
- benchmark scripts that summarize tensor sizes, virtual bond dimensions, build times, and dense-validation errors;
- a Max-Cut cost-operator example using diagonal Z_iZ_j terms;
- a long-range $XX + YY + Z$ Hamiltonian example using dense Pauli operators;
- a quantum clock Hamiltonian example using generalized Pauli operators for qudits;
- a user-defined Heisenberg Hamiltonian example that supplies its own local-operator store and symbolic-term generator;
- unit tests for token parsing, grouping, center selection, tensor construction, Hamiltonian helpers, and the public front-end.

2.3. Automata-inspired TN construction

Based on [3], the MPO construction implemented in this software is based on the idea that different symbolic operator strings often share common partial structures. Instead of treating every term independently, the software identifies repeated prefixes and suffixes among the symbolic strings and reuses them when constructing the virtual bonds of the MPO. In this sense, the virtual bonds act as cached partial histories of the symbolic operator strings.

Formally, let $T = \{t_1, \dots, t_m\}$ be a set of symbolic operator strings, where each string $t_k = (s_{k,1}, \dots, s_{k,N})$ contains one local token[1] per site. For a candidate split position, `autotn` groups terms by their left prefixes $(s_{k,1}, \dots, s_{k,i})$ and right suffixes $(s_{k,i+1}, \dots, s_{k,N})$. These equivalence classes become virtual states.

At site i, the tensor records the allowed transitions between virtual states. A transition is included when at least one term in T connects the corresponding partial operator strings through its local token at site i.

As a simple example, consider a four-site diagonal operator containing all pairwise Z operations,

$$\sum_{1 \le i < j \le 4} Z_i Z_j. \tag{2}$$

With the local alphabet $\mathcal{A} = \{I, Z\}$, the corresponding symbolic strings are

$$T = \{ZZII,\ ZIZI,\ ZIIZ,\ IZZI,\ IZIZ,\ IIZZ\}. \tag{3}$$

In this example $N = 4$ is the number of sites, $m = 6$ is the number of symbolic strings, and $|\mathcal{A}| = 2$ is the number of different local tokens used. For instance, $t_2 = ZIZI$ denotes the local-token tuple (Z, I, Z, I), corresponding to $Z_1 I_2 Z_3 I_4$. Several of these strings share common partial patterns. For example, `ZZII`, `ZIZI`, and `ZIIZ` all begin with the same first token, `Z`, indicating that the first site already carries a non-identity operator in those terms. Similarly, strings such as `ZIIZ`, `IZIZ`, and `IIZZ` share suffix patterns that end with `Z`, indicating that the last site participates in the corresponding interaction. More generally, common prefixes encode identical partial structures from the left of the chain, while common suffixes encode identical partial structures from the right. `autotn` reuses these repeated partial structures as virtual states of the MPO instead of treating each symbolic string as an independent term.

This is the sense in which the implementation is automata-inspired. The prefix and suffix classes play a role analogous to states in a finite-state description: they encode the partial symbolic configurations identified before and after a given site. These states are then associated with the virtual bonds of the MPO, allowing repeated symbolic structure to be reused during tensor construction. In `autotn`, this automata-inspired formulation is implemented through an internal prefix/suffix construction that compiles symbolic operator strings into concrete TN arrays. The following schematic summarizes the construction:

1. Generate symbolic local strings and optional coefficient-vector references.
2. For each candidate center, estimate the number of prefix/suffix states and tensor entries.
3. Select the center according to the requested strategy, such as `default` or `min_elements`.
4. Build only the selected tensor network by resolving symbolic tokens through `store_matrix`.

A relevant feature of this workflow is that candidate split points are scored by estimating tensor shapes before construction. The final tensor network is built only for the selected candidate. This is useful in research settings where one wants to inspect memory-related metrics without materializing every possible tensor network.

For center selection, `default` evaluates only the documented default split $\mu_0 = \max(0, \lfloor N/2 \rfloor - 1)$. The `min_elements` strategy scores candidate split positions using exact tensor-shape estimates. For dense local-operator models, the default candidate set is exhaustive over $\mu = 0, \dots, N-1$. For diagonal models with $N > 2$, the default candidate set uses the interior positions $\mu = 1, \dots, N-2$, whereas $N \le 2$ uses all positions. In all cases, the default split μ_0 is explicitly included in the candidate list.

The candidate search is shape-based rather than construction-based: `min_elements` does not materialize a full TN for every candidate split. Instead, it regenerates the symbolic terms for each candidate when needed, estimates the resulting tensor shapes from the prefix/suffix grouping structure, and constructs only the TN associated with the selected split.

`min_elements` selects the candidate with the smallest estimated total tensor-entry count. Ties are resolved by smaller $D_{\max}$, then smaller distance to μ_0, then smaller split index. Since μ_0 is among the candidates and the same tensor-element metric is used for scoring, `min_elements` cannot select a construction with more estimated tensor elements than `default`; it may still increase build time because several candidates are evaluated before building the final TN.

3. Illustrative examples

This section shows how `autotn` is used in practice. The aim is to move from the internal description of the package to concrete examples that a reader can run, inspect, and adapt. Each example follows the same basic idea: a model is defined through its numerical parameters, the symbolic representation is generated by the corresponding model class, and `TNGenerator` builds the tensor network together with useful diagnostic information. For computationally tractable instances, the resulting tensors are also contracted and compared against an independent dense reference, so that the examples illustrate not only how to use the software, but also how its output can be checked. This workflow is intentionally simple, since the goal is to make the construction process transparent before considering larger instances where dense validation is no longer practical.

3.1. Model/generator workflow

The executable examples use the same high-level workflow: construct or load model parameters, wrap them in a model object, build the tensor network with `TNGenerator`, and validate in computationally tractable cases against a dense reference function from `autotn.references`. The following abbreviated example follows the

[1] In this context, a token denotes the symbolic label of the local object assigned to one site in an operator string. For example, in the string `ZIZI`, the tokens are `Z`, `I`, `Z`, and `I`, corresponding to the local operator product $Z_1 I_2 Z_3 I_4$.

implemented $XX + YY + Z$ script in `examples/xxyy_hamiltonian_usage_example.py`.

```
import numpy as np
from autotn import TNGenerator
from autotn.operator import dense_operator_from_tn
from autotn.hamiltonians import
   generate_symmetric_couplings_and_fields
from autotn.models import XXYYZModel
from autotn.references import xxyyz_dense_reference

J, h = generate_symmetric_couplings_and_fields(n_sites=4,
    seed=123)
model = XXYYZModel(J=J, h=h)
tn, info = TNGenerator(model).build()

contracted = dense_operator_from_tn(tn)
reference = xxyyz_dense_reference(J, h)
assert np.allclose(contracted, reference)

print(info.selected_middle)
print(info.max_bond_dimension)
print(info.total_tensor_elements)
```

The example can be read line by line as a compact version of the full workflow. **Lines 1–6** import the objects needed to run the construction and validation: `numpy` provides the numerical comparison, `TNGenerator` builds the tensor network, `dense_operator_from_tn` contracts it when the instance is computationally tractable, and the remaining imports provide the parameter generator, model wrapper, and independent dense reference. **Line 8** creates the numerical parameters of the problem, namely the couplings J and fields h, with a fixed seed so that the example is reproducible. **Line 9** wraps these arrays in an `XXYYZModel`, which exposes the local operator store and symbolic terms expected by `autotn`. **Line 10** is the actual TN construction: `TNGenerator(model).build()` returns the tensor list `tn` and a metadata object `info`. **Lines 12–14** perform the dense check on a computationally tractable instance. The generated tensor network is contracted, an independent dense Hamiltonian is built directly from J and h, and both are compared with `np.allclose`. Finally, **Lines 16–18** print diagnostic information: the selected center, the maximum virtual bond dimension, and the total number of tensor entries.

The returned `tn` is a list of NumPy arrays. For this dense local-operator example, each tensor has four axes. The `info` object records the selected split point and shape metrics. The validation line compares the contracted TN operator against an independently constructed dense reference from `autotn.references`. This reference implements the Hamiltonian formula directly from J and h, rather than reusing the symbolic token generator.

3.2. Currently implemented Hamiltonian examples

Four executable examples are currently included in the repository. Table 1 summarizes their scripts, model wrappers or script-local model definitions, backends, and dense-reference functions. They differ in local dimension, backends, and dense-reference formulas, but all use the model/generator interface.

The first three examples use packaged model wrappers, while the Heisenberg example below shows that the same interface can also be supplied directly by user code. The implemented operator for each model is stated alongside its description below.

A brief description of each example is provided below.

1. **Max-Cut cost operator.** The example generates a weighted adjacency matrix, wraps it in `MaxCutModel`, and builds a diagonal TN through `TNGenerator`. It compares the default center with the model's size-oriented center strategy and checks the optimized tensor network against `maxcut_dense_reference`. The implemented operator is

$$C_{\text{MaxCut}}(W) = \sum_{1\le i<j\le N} W_{ij} Z_i Z_j.$$

   Here, W_{ij} is the weight associated with the pair (i, j), and unlisted sites are acted upon by the identity. This is the implemented Ising form of the weighted cut objective, which differs from the usual cut value by an affine transformation.
2. **Long-range $XX + YY + Z$ Hamiltonian.** The script `examples/xxyy_hamiltonian_usage_example.py` generates reproducible J and h parameters, builds a dense local-operator tensor network through `XXYYZModel` and `TNGenerator`, prints symbolic term metadata, and validates the contracted operator against `xxyyz_dense_reference`. The implemented Hamiltonian is

$$H_{\text{XXYYZ}}(J, h) = \sum_{1\le i<j\le N} J_{ij}(X_i X_j + Y_i Y_j) + \sum_{i=1}^{N} h_i Z_i.$$

   Here, J_{ij} denotes the coupling between sites i and j, h_i is the local longitudinal field, and X, Y, and Z are Pauli matrices. Unlisted sites are acted on by the identity.
3. **Quantum clock Hamiltonian.** The script `examples/quantum_clock_model_usage_example.py` constructs long-range qudit clock Hamiltonians for several N and q values. The implementation treats $q = 2$ as a special case in which dagger-paired terms collapse, reports the expected symbolic term count, and validates computationally tractable dense operators against `quantum_clock_dense_reference`. The implemented Hamiltonian is:

$$H_{\text{clock}}(J, h; q) = -\sum_{1\le i<j\le N} J_{ij}\left(Z_i^{\dagger} Z_j + Z_i Z_j^{\dagger}\right) - \sum_{i=1}^{N} h_i \left(X_i + X_i^{\dagger}\right).$$

   Here, X is the q-dimensional cyclic-shift matrix, while the clock matrix and phase are defined as

$$Z = \text{diag}(1, \omega, \ldots, \omega^{q-1}), \qquad \omega = \exp(2\pi \mathrm{i}/q).$$

   The lowercase operator tokens used in the implementation denote daggered operators. When $q = 2$, dagger-paired terms are combined because $X = X^{\dagger}$ and $Z = Z^{\dagger}$, producing the corresponding factors of two in the generated coefficients. Unlisted sites are acted on by the identity.
4. **User-defined Heisenberg Hamiltonian.** The script `examples/user_heisenberg_model_example.py` defines a nearest-neighbor

Table 1
Executable examples are included in the repository. The dense reference implementations are independent of the symbolic token generators and are used to validate the resulting tensor networks for computationally tractable instances.

Example script	Model interface	Backend	Dense reference
`examples/maxcut_usage_example.py`	`MaxCutModel`	Diagonal vectors	`maxcut_dense_ reference(W)`
`examples/xxyy_hamiltonian_usage_example.py`	`XXYYZModel`	Dense local matrices	`xxyyz_dense_ reference(J,h)`
`examples/quantum_clock_model_usage_example.py`	`Quantum ClockModel`	Dense local matrices	`quantum_clock_dense_reference (J,h,q)`
`examples/user_heisenberg_model_example.py`	script-local `User Heisenberg Model`	Dense local matrices	script-local dense reference

Heisenberg chain directly in the example file, including its local Pauli-operator store, its `coefficient_vector`, and its `symbolic_terms` method. The scalar couplings and site-dependent fields are mapped to coefficient indices by `build_coefficient_vector`, and each generated XX, YY, ZZ, or Z operator string receives the appropriate weighted token at the selected `middle` position before validation against a script-local dense reference. The implemented Hamiltonian is

$$H_{\mathrm{Heis}} = \sum_{i=1}^{N-1} \left(J_x X_i X_{i+1} + J_y Y_i Y_{i+1} + J_z Z_i Z_{i+1} \right) + \sum_{i=1}^{N} h_i Z_i.$$

The snippet below shows how this expression is converted into symbolic strings. Coefficients are stored once in `coefficient_vector`; `V-k_A` multiplies operator `A` by coefficient index `k`. The full script provides `store_matrix` and builds the model with the `min_elements` center strategy.

```
coeffs, (jx, jy, jz, h_ids) = build_coefficient_vector(Jx, Jy, Jz, h)
terms, base = {}, ["I"] * N
def add(tokens, coeff_id):
    tokens[middle] = f"V-{coeff_id}_{tokens[middle]}"
    terms[len(terms)] = tokens
for site in range(N - 1):
    for token, coeff_id in (("X", jx), ("Y", jy), ("Z", jz)):
        pair = base.copy()
        pair[site] = pair[site + 1] = token
        add(pair, coeff_id)
for site, coeff_id in enumerate(h_ids):
    field = base.copy()
    field[site] = "Z"
    add(field, coeff_id)
symbolic_input = SymbolicTNInput(terms=terms, coefficient_vector=coeffs)
```

4. Validation and benchmark results

The repository includes two benchmark workflows. The build benchmark measures construction time and structural TN metrics without dense validation. The validation benchmark contracts computationally tractable generated networks and compares them with dense Hamiltonians built by separate reference functions in `autotn.references`. These functions implement the model equations directly from the numerical parameters, such as W, J, h, and q, and do not use symbolic token lists or TN construction. This validation is designed to catch mismatches between symbolic rules and generated TN arrays, which is precisely the class of implementation errors that often occur when MPOs or related operator networks are assembled manually.

Table 2 lists the configurations used in the two benchmarks. We use the same procedure to generate random instances in both cases. The main difference is that the construction benchmark only builds the TN, whereas the validation benchmark also builds the corresponding dense operator. Since the dense representation grows rapidly with the Hilbert-space dimension, we perform this comparison only for cases that remain computationally manageable.

Each benchmark run also records the symbolic metadata introduced by the front-end architecture. The fields `n_terms` and `n_coefficients` are not performance metrics by themselves: they document the symbolic workload being compiled and make it explicit when several operator strings share entries in the same coefficient vector. They provide provenance for the structural measurements reported below, while the visible benchmark summaries focus on total tensor elements, $D_{\max}$, construction time, and dense-validation error. The benchmark results reported below summarize the build-size and build-time trade-off in Table 3 and Figs. 2–4, followed by the dense validation coverage in Table 4 and the error values reported there. These results are intended as reproducibility and reliability checks for the current implementation, not as hardware-independent performance claims.

The current build benchmark contains 4500 construction runs, corresponding to 30 repeated timings per tested configuration and center strategy. It compares the default center with the `min_elements` strategy on matched model configurations. In this comparison, an "improvement" means a reduction in the total number of tensor entries, which is a structural proxy for memory footprint and TN size. It does not mean that the construction is faster. The `min_elements` strategy evaluates candidate split points before building the final tensor network, so it can reduce tensor size while still increasing build time. Table 3 summarizes this size-time trade-off.

The reported benchmark data were also checked at the individual matched-configuration level. Across the 75 default-versus-`min_elements` comparisons, `min_elements` reduced the total tensor-element

Table 3
Summary of the `min_elements` strategy relative to the default center. Element ratios are average ratios of mean total tensor elements over matched configurations. Build-time ratios are average ratios of median build times. Improved, tied, and worse count matched configurations for which `min_elements` reduced, preserved, or increased the total tensor size. The total differs across model families because quantum clock instances are evaluated for three local dimensions, $q = 2, 3, 4$, while Max-Cut and XXYYZ have no q sweep.

Model	Element ratio	Build-time ratio	Improved	Tied	Worse	Total
Max-Cut	0.992	1.63	6	9	0	15
XXYYZ	1.000	1.41	0	15	0	15
Quantum Clock	0.967	1.48	12	33	0	45

Table 2
Benchmark configurations used for construction measurements and dense validation.

Workflow and model	Sizes N	q / local space	Instance parameters	Protocol and reported quantities
Build: Max-Cut	8, 10, 12, 16, 20	Diagonal binary variables	Seeds 0, 1, 2; random weighted graph with edge probability 0.5; integer weights in $[1, 10]$	`default` and `min_elements`; 5 warm-up builds and 30 timed repeats; reports time, split, n_{terms}, $n_{\mathrm{coefficients}}$, $D_{\max}$, total entries, memory, and candidate centers
Build: $XX + YY + Z$	$\{2, 3, 4, 5, 6\}$	Qubits	Seeds 0, 1, 2; symmetric couplings J_{ij} and local fields h_i sampled uniformly from $[-1, 1]$	Same build protocol and metrics
Build: quantum clock	$\{2, 3, 4, 5, 6\}$	$q = \{2, 3, 4\}$	Seeds 0, 1, 2; symmetric couplings J_{ij} and local fields h_i sampled uniformly from $[-1, 1]$	Same build protocol and metrics
Dense validation: Max-Cut	4, 6, 8, 10, 12	Diagonal binary variables	Same seed and graph-generation rules	One validation per configuration and center strategy; dense-vector checks capped at dimension 4096; reports exact, skipped, and failed counts plus maximum error
Dense validation: $XX + YY + Z$	$\{2, \ldots, 10\}$	Qubits	Same seed and parameter-generation rules	One validation per configuration and center strategy; dense-matrix checks capped at dimension 1024; reports exact, skipped, and failed counts plus maximum error
Dense validation: quantum clock	$\{2, \ldots, 10\}$	$q = \{2, 3, 4\}$	Same seed and parameter-generation rules	One validation per configuration and center strategy; dense-matrix checks capped at dimension 1024; reports exact, skipped, and failed counts plus maximum error

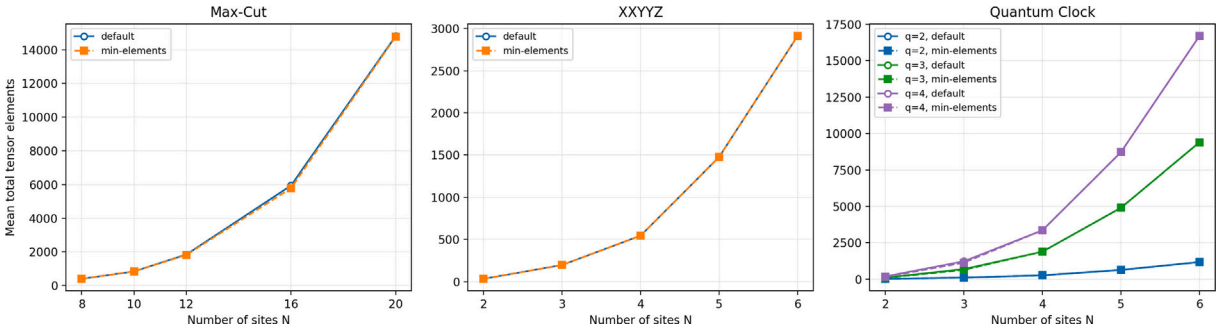


Fig. 2. Mean total tensor elements for the benchmarked models. The panels separate Max-Cut, XXYYZ, and quantum clock instances. For XXYYZ, the two center strategies overlap because they produce identical tensor sizes in the tested range.

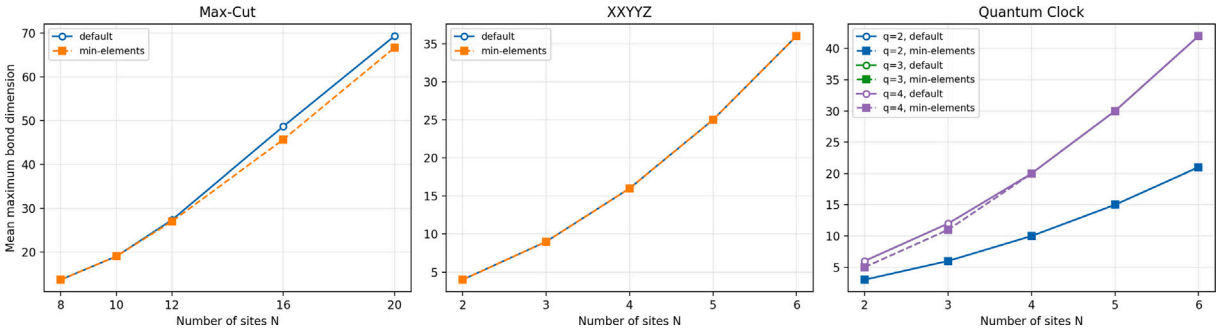


Fig. 3. Mean maximum virtual bond dimension $D_{\max}$ for the same build benchmark. The `min_elements` strategy is primarily a total-size criterion, so $D_{\max}$ should be interpreted as a diagnostic rather than as the optimized objective.

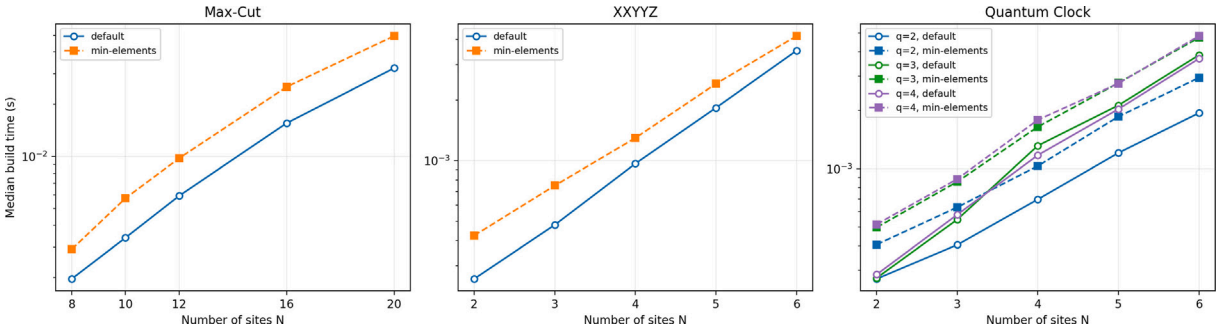


Fig. 4. Median TN build time. The vertical axis is logarithmic. The `min_elements` strategy has additional overhead because it evaluates candidate centers before building the selected tensor network.

Table 4
Dense-validation summary for the current benchmark data. Skipped cases correspond to configurations above the reported dense-reference Hilbert-space limits: dimension 4096 for diagonal dense-vector checks and dimension 1024 for dense operator-matrix checks.

Model	Passed	Skipped	Failed	Largest validated cases	Max. abs. error
Max-Cut	30	0	0	$N = 12$, dim = 4096	0.0
XXYYZ	54	0	0	$N = 10$, dim = 1024	0.0
Quantum Clock	108	54	0	$q = 2 : N = 10, \dim = 1024$; $q = 3 : N = 6, \dim = 729$; $q = 4 : N = 5, \dim = 1024$	1.95×10^{-14}

count in 18 cases, tied the default construction in 57 cases, and was worse in 0 cases.

The interpretation is deliberately limited. For Max-Cut, the element ratio is 0.992, indicating a very small average size reduction: six matched configurations improve and nine are unchanged. For XXYYZ, the ratio is 1.000 and all matched configurations are tied, so the alternative center selection does not reduce tensor size for this model family in the tested range. The clearest structural reduction appears for the quantum clock cases, where the mean element ratio is 0.967 and 12 of 45 matched configurations improve. In all three model families, however, the median construction time increases by roughly 1.4×–1.6×, because the size-oriented strategy spends extra time estimating and ranking candidate centers. Thus, `min_elements` should be described as a tensor-size heuristic rather than as a general performance optimization.

Table 5
Current feature scope. The software is designed around operator construction and validation, not around solving TN optimization problems.

Supported in the current repository	Not supported in the current repository
Symbolic local-token descriptions of operator strings	General symbolic algebra or expression simplification
Diagonal local-vector TN	DMRG, MPS optimization, or variational solvers
Dense local-operator TN	Automatic fermionic sign handling or second-quantized algebra
Weighted tokens through coefficient-vector lookup	Large-scale dense validation beyond tractable Hilbert-space dimensions
Automatic center selection by estimated tensor size	Dataset management or full experiment orchestration
Dense reference validation for computationally tractable instances	Replacement of mature TN solver libraries

Also, because the strategy optimizes total tensor elements rather than only $D_{\max}$, occasional increases in maximum bond dimension are possible when they reduce the total tensor size. The detailed scaling plots separate these effects: Fig. 2 shows the mean total tensor elements, Fig. 3 reports the corresponding maximum virtual bond dimensions, and Fig. 4 shows the median construction time.

Dense validation addresses a different question: correctness rather than improvement. It was run for 246 cases across both center strategies. Of these, 192 passed and 54 were skipped because their Hilbert-space dimension was above the dense-reference limit configured for the corresponding backend. Here this dimension is the number of basis configurations that the dense reference would need to represent. For the reported run, the limit was dimension 4096 for diagonal dense-vector validation and dimension 1024 for dense operator-matrix validation. No dense-validation failures were observed. Table 4 summarizes the current validation coverage. Max-Cut and XXYYZ match exactly within the implemented floating-point comparison. The complex-valued quantum clock model reaches a maximum absolute deviation of 1.95×10^{-14}, consistent with numerical round-off from dense contraction and complex arithmetic. This validation does not prove asymptotic performance or cover arbitrarily large systems, but it directly verifies that the generated TN operators agree with dense Hamiltonians constructed directly from W, J, h, and q, independently of the symbolic token lists.

5. Impact

The main contribution of `autotn` is a reproducible path from symbolic local operator rules to exact TN arrays. It reduces the engineering barrier between model specification and validated TN operator construction. This is useful in research workflows where new Hamiltonian or cost-function encodings must be prototyped, inspected, and compared without repeatedly hand-coding MPOs.

`autotn` separates three concerns that are often mixed: symbolic operator generation, TN construction, and exact validation on computationally tractable instances. This separation helps users identify whether an error comes from the model rule, the tensor construction, or later numerical use of the operator. The structural diagnostics also expose how grouping and center-selection decisions affect $D_{\max}$, memory-related tensor sizes, and build-time overhead. As a result, the package provides a reusable construction and validation layer that can support later integration with other TN solvers and benchmarking pipelines.

6. Conclusions and future work

`autotn` implements an automata-inspired approach for constructing exact TN representations from symbolic local operator strings. The current repository demonstrates diagonal and dense local-operator compilation, automatic split-point selection, structural diagnostics, and validation against dense references on computationally tractable instances. The design is intentionally compact: users define local arrays and symbolic operator rules, while the software builds and checks the corresponding TN representation.

Table 5 summarizes the practical boundary of this first release. The supported column reflects the part of the workflow that the package currently owns: turning finite symbolic local-token descriptions into concrete NumPy TN arrays and checking them on computationally tractable instances. The unsupported column is equally important because it separates this construction layer from tasks that belong to broader TN frameworks or symbolic algebra systems. In this sense, the current software is best understood as a focused operator-generation and validation component rather than a complete simulation environment.

Planned work follows directly from this boundary: improving documentation of the symbolic front-end, extending benchmark coverage to larger and more diverse models, and integrating the generated TN with solver workflows. Further Hamiltonian examples beyond those already provided should be added only once they are implemented, tested, and included in the validation pipeline.

CRediT authorship contribution statement

Aitor Morais: Writing – original draft, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Izaskun Oregi:** Writing – review & editing, Validation, Supervision, Investigation, Conceptualization. **Iker Pastor:** Funding acquisition. **Eneko Osaba:** Writing – review & editing, Validation, Supervision, Investigation, Conceptualization.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of the manuscript, the authors used ChatGPT (OpenAI) as an assistive tool to review the manuscript for clarity and consistency, and to help review and improve portions of the software code originally developed by the authors. All suggestions and code modifications were manually reviewed, tested, and validated by the authors.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Aitor Morais and Iker Pastor acknowledge partial funding for Aitor Morais's doctoral research at the University of Deusto, within the D4 K (Deusto for Knowledge) team on applied artificial intelligence and quantum computing technologies.

References

[1] Lucas A. Ising formulations of many NP problems. Front Phys 2014;2:74887.
[2] Schollwöck U. The density-matrix renormalization group in the age of matrix product states. Ann Phys 2011;326(1):96–192. https://doi.org/10.1016/j.aop.2010.09.012
[3] Crosswhite GM, Bacon D. Finite automata for caching in matrix product algorithms. Phys Rev A 2008;78(1):012356. https://doi.org/10.1103/PhysRevA.78.012356
[4] Hubig C, McCulloch IP, Schollwöck U. Generic construction of efficient matrix product operators. Phys Rev B 2017;95(3):035129. https://doi.org/10.1103/PhysRevB.95.035129
[5] Orús R. A practical introduction to tensor networks. Ann Phys 2014;349:117–58. https://doi.org/10.1016/j.aop.2014.06.013
[6] Pirvu B, Murg V, Cirac JI, Verstraete F. Matrix product operator representations. New J Phys 2010;12(2):025012.
[7] Haegeman J, Verstraete F. Diagonalizing transfer matrices and matrix product operators: a medley of exact and computational methods. Annu Rev Condens Matter Phys 2017;8(1):355–406.
[8] Paeckel S, Köhler T, Manmana SR. Automated construction of u(1)-invariant matrix-product operators from graph representations. SciPost Physics 2017;3(5):035. https://doi.org/10.21468/SciPostPhys.3.5.035
[9] Ren J, Li W, Jiang T, Shuai Z. A general automatic method for optimal construction of matrix product operators using bipartite graph theory. J Chem Phys 2020;153(8):084118. https://doi.org/10.1063/5.0018149

[10] Fishman M, White SR, Stoudenmire EM. The ITensor software library for tensor network calculations. SciPost Physics Codebases 2022;4. https://doi.org/10.21468/SciPostPhysCodeb.4

[11] Hauschild J, Pollmann F. Efficient numerical simulations with tensor networks: tensor network Python (TeNPy). SciPost Physics Lecture Notes 2018;5. https://doi.org/10.21468/SciPostPhysLectNotes.5

[12] Gray J. quimb: a Python package for quantum information and many-body calculations. J Open Source Softw 2018;3(29):819. https://doi.org/10.21105/joss.00819

[13] McClean JR, Rubin NC, Babbush R, Bacon D, Bushnell C, Boixo S, McClain C, Neven H. OpenFermion: the electronic structure package for quantum computers. Quantum Science and Technology 2020;5(3):034014. https://doi.org/10.1088/2058-9565/ab8ebc