1 Features
  1.1 Installing ................................................................. 3
  1.2 Tips, Tricks, and Pitfalls .............................................. 5
  1.3 FAQ ................................................................. 6
  1.4 Full API documentation .............................................. 6

Python Module Index ...................................................... 23

Index ................................................................. 25
Welcome to **dynamite**, which provides a simple interface to fast evolution of quantum dynamics and eigensolving. Behind the scenes, dynamite uses the PETSc/SLEPc implementations of Krylov subspace exponentiation and eigensolving.

For illustrative examples, check out dynamite on [GitHub](https://github.com)! The techniques implemented by dynamite fill a niche for numerical quantum simulation. DMRG methods are very fast but their speed depends on entanglement, preventing evolution past moderate time scales for systems that become highly entangled. Exact diagonalization allows for evolution to arbitrarily long times, but is quite limited in Hilbert space size. dynamite is best at evolving for moderate time scales (perhaps $\sim 10^4 \cdot 1/|J|$) on moderate size Hilbert spaces (up to $\sim 30$ spins or so).

**Note:** dynamite is in beta! You may find bugs. When you do, please submit them on the [GitHub Issues](https://github.com/issues) page! Additionally, you may want to check you are getting correct answers by comparing a small system to output from a different method.
CHAPTER 1

Features

- Easy building of spin chain Hamiltonians through Python
- Performance-critical code written in C, giving speed comparable to pure C implementation
- Underlying PETSc/SLEPc libraries supply fast algorithms for matrix exponentiation and eigensolving
- Options such as shell matrices provide customizability and access to extremely large Hilbert spaces

1.1 Installing

**Note:** dynamite is written for Python >=3.6! You may need to install an appropriate version first if you don’t already have it.

1.1.1 Download dynamite

```bash
git clone https://github.com/GregDMeyer/dynamite.git
```

Dynamite is built on the PETSc and SLEPc packages. The next step is to install those.

1.1.2 Build PETSc

To build PETSc in your working directory, as per the download page, do the following. There is a configuration script that comes with dynamite which should help:

```bash
git clone https://gitlab.com/petsc/petsc.git petsc
cd petsc
git checkout tags/v3.15.0
python <dynamite directory>/petsc_config/complex-opt.py
```
Note that you may want to adjust some of the build options. Just take a look at the script and modify as desired. There are also a couple other scripts in that directory for debug builds (if you will be modifying dynamite) and CUDA support.

If all goes well, configure will tell you to run a make command. Copy the command and run it. It should look like: `make PETSC_DIR=<your_petsc_directory> PETSC_ARCH=complex-opt all`

### 1.1.3 Building SLEPc

Before installing SLEPc, make sure to set environment variables describing your new PETSc installation:

For bash shell: `export PETSC_DIR=<petsc_dir>; export PETSC_ARCH=complex-opt`

Now download and install SLEPc:

```
git clone https://gitlab.com/slepc/slepc.git slepc
cd slepc
git checkout tags/v3.15.0
./configure
```

If it configures correctly, it will output a make command to run. Copy and paste that, and run it. It should look like: `make SLEPC_DIR=$PWD PETSC_DIR=<petsc_dir> PETSC_ARCH=complex-opt`

### 1.1.4 Building dynamite

Dynamite requires Python 3, as well as some packages you can install with pip. These are listed in `requirements.txt` in the dynamite root directory. Two of the packages, petsc4py and slepc4py, are Python wrappers for PETSc and SLEPc. Before you install them, make sure PETSC_DIR and PETSC_ARCH environment variables are still set from the above exports (or re-set them). You should also set SLEPC_DIR with `export SLEPC_DIR=<your_slepc_installation_directory>`. Then, you can install everything by just running

```
cd dynamite
pip install -r requirements.txt
```

Note: For some reason, this step sometimes fails because pip tries to install petsc4py before cython, even though cython is required by petsc4py. To fix this, simply run `pip install cython` and then try again with `requirements.txt`.

Note: When using pip with sudo, you need to pass the -E flag to sudo to preserve the environment variables (PETSC_DIR etc.).

I suggest using a virtual environment, to keep all of the packages tidy.

Finally, install dynamite:

```
pip install ./ # you may want sudo with pip
```

Now you should be all set to use dynamite! If you want to work on the dynamite source code, or just easily pull updates from GitHub, you might want to do `pip install -e ./` to keep the source files in-place.

Note: Don’t try to do pip install dynamite! There is a totally unrelated package on PyPI by that name.
1.2 Tips, Tricks, and Pitfalls

1.2.1 Pitfalls (TL;DR)

- Beware of “nondeterministic” code when running in parallel!
  - Making calls to e.g. “numpy.random.rand()” will give different values on each process. If you use this to build your Hamiltonian, you will not have a consistent operator across your different processes! If you need random numbers, make sure to seed them with the same value everywhere.
  - An especially sneaky example is iterating through dictionaries: since they are unordered, if you do for key, value in d.items():, you may get items in a different order and your processes will not be running the same code. One solution is to set PYTHONHASHSEED=0 in the environment before starting your Python interpreter; this disables the randomization in the hash function.

- It really is useful to read the SLEPc (and PETSc) Users’ Manual!

1.2.2 Parallelism

PETSc and SLEPc are built to leverage massively parallel computing. They use the MPI (message-passing interface) framework to communicate between processes. Accordingly, dynamite should be run with MPI. To do so (the -n flag specifies the number of processes):

```
mpirun -n 4 python3 solve_all_the_things.py
```

Or, if you have MPI installed in an unusual way (e.g. PETSc installed it for you), you may want:

```
$PETSC_DIR/bin/petscmpiexec -n 4 python3 solve_quantum_gravity.py
```

1.2.3 Matrix-free matrices

“Matrix-free” matrices (known in dynamite and PETSc as “shell” matrices) save significantly on memory usage and can also sometimes speed things up. Instead of storing the entire matrix in memory, they compute matrix elements on-the-fly when they are needed. When using shell matrices, the only significant memory usage is the storage of the state vector (and any other vectors used in the evolution or eigensolve computations).

See dynamite.operators.Operator.shell for details.

1.2.4 Jupyter Notebook Integration

dynamite integrates well with Jupyter Notebooks, outputting the form of operators in TeX representation inline. However, getting MPI set up under Jupyter is a bit of a hassle, so it’s best for small computations on one process.

1.2.5 Interacting with PETSc/SLEPc

The underlying petsc4py matrix for any operator is accessible with dynamite.operators.Operator.get_mat(). For states, the petsc4py vector is dynamite.states.State.vec. Arbitrary functions from petsc4py can be called through this interface. The documentation is not too extensive for petsc4py and slepc4py, but it is inferred easily from the C interface. For example, the C function MatMult() is implemented as a member function of the Python petsc4py.PETSc.Mat() class: one would just do my_matrix.mult(in_vec, result_vec).
C programs using PETSc and SLEPc can take options at runtime that modify how the libraries run. These options can
be passed to dynamite as well. It is accomplished by using dynamite.config.initialize(). An example:
to change the size of the Krylov subspace used by SLEPc’s matrix exponential, one would do

```python
from dynamite import config
config.initialize( ['-mfn_ncv', '40']
```

### 1.2.6 GPU Support

It is possible to run dynamite computations on GPUs, and it is generally quite fast. However this functionality is still
experimental; make sure to check your results!

The basic steps are the following:

- Find a machine with an Nvidia GPU and CUDA version > 11.0
- Build PETSc/SLEPc using the `cuda-opt.py` configuration script (in the `petsc_config` directory of dy-
namite)
- Build dynamite with PETSC_ARCH environment variable set to `cuda-opt`
- Call `config.initialize(gpu=True)` at the beginning of your script

A drawback of GPUs is their limited memory that restricts possible system sizes; to get to bigger system sizes try
using shell matrices on the GPU (by setting `config.shell = True`).

### 1.3 FAQ

**Why is dynamite crashing with some message about MPI INIT?**

Are you running with `mpirun`? PETSc and SLEPc are sometimes unhappy if they are run outside of MPI (see
Parallelism).

**I tried to run dynamite in a Jupyter Notebook and my kernel died on import. Why?**

This seems to be a result of the same situation as above. Running MPI in a notebook to keep the libraries happy (and
allow use of more than one process) is possible and pretty cool—just takes a bit of work to set up.

**I am so tired of setting the size of all my matrices to the same value!**

That wasn’t a question. But anyway, there is an easy way to globally set a default value for \( L \). Before you start building
any operators:

```python
from dynamite import config
config.L = 24  # or whatever you want
```

There are other global configuration options, too. See the full documentation for details.

### 1.4 Full API documentation

#### 1.4.1 dynamite.operators
Single-site operators

These operators apply to a single spin, identified by an index passed to their constructor. The default index is 0. It is implied that they are tensored with the identity on every other site in the spin chain. Explicitly, a single spin operator $O$ here has the form $I_0 \otimes I_1 \otimes \ldots \otimes O_i \otimes \ldots \otimes I_L$, where $i$ is the index passed in the constructor and $L$ is the length of the spin chain.

dynamite.operators.$\texttt{sigmax}(i=0)$
The Pauli $\sigma_x$ operator on site $i$.
dynamite.operators.$\texttt{sigmay}(i=0)$
The Pauli $\sigma_y$ operator on site $i$.
dynamite.operators.$\texttt{sigmaz}(i=0)$
The Pauli $\sigma_z$ operator on site $i$.
dynamite.operators.$\texttt{sigma_plus}(i=0)$
The $\sigma_+ = \sigma_x + i\sigma_y$ operator.

\begin{equation}
\text{Note: } \sigma_+ = \begin{pmatrix} 0 & 2 \\ 0 & 0 \end{pmatrix}, \text{ so } S_+ = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \frac{1}{2}\sigma_+
\end{equation}


dynamite.operators.$\texttt{sigma_minus}(i=0)$
The $\sigma_- = \sigma_x - i\sigma_y$ operator.

\begin{equation}
\text{Note: } \sigma_- = \begin{pmatrix} 0 & 0 \\ 2 & 0 \end{pmatrix}, \text{ so } S_- = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} = \frac{1}{2}\sigma_-
\end{equation}

dynamite.operators.$\texttt{identity}()$
The identity operator.
dynamite.operators.$\texttt{zero}()$
The zero operator—equivalent to a matrix of all zeros.

Sums and Products

dynamite.operators.$\texttt{op_sum}(\text{terms, nshow=3})$
A sum of several operators. This object can be used in a couple ways. All of the following return the exact same object, $\sigma_0^x + \sigma_0^y$.

\begin{verbatim}
sigmax() + sigmay()
op_sum([sigmax(), sigmay()])
\end{verbatim}

\begin{verbatim}
for s in [sigmax, sigmay]
\end{verbatim}

Parameters

- $\texttt{terms (list)}$ – A list of operators to sum
- $\texttt{nshow (int, optional)}$ – The number of terms to show in the string representations before adding an ellipsis.

dynamite.operators.$\texttt{op_product}(\text{terms})$
A product of several operators. Called in same way as $\texttt{op_sum()}$. For example:
>>> sigmax() * sigmay() == op_product([sigmax(), sigmay()])
True

Parameters `terms (list)` – A list of operators to multiply

Translations

The following two classes translate the given operator `op` along the spin chain, and take the sum or product of the results.

dynamite.operators.index_sum(`op`, `size=None`, `start=0`, `boundary='open'`)
Duplicate the operator onto adjacent sites in the spin chain, and sum the resulting operators. In most cases, `op` should have support on site 0 (and possibly others).

See the examples for more information.

Parameters

- `op (Operator)` – The operator to translate along the spin chain.
- `size (int, optional)` – The size of the support of the resulting operator. For open boundary conditions, the number of terms in the sum may be smaller than this. If not provided, defaults to the value of `Operator.L`.
- `start (int, optional)` – The site for the first operator in the sum.
- `boundary (str, optional)` – Whether to use ‘open’ or ‘closed’ boundary conditions. When `op` has support on more than one site, this determines whether the last few terms of the sum should wrap around to the beginning of the spin chain.

dynamite.operators.index_product(`op`, `size=None`, `start=0`)
Duplicate the operator onto adjacent sites in the spin chain, and multiply the resulting operators together. In most cases, `op` should have support on site 0 (and possibly others).

Parameters

- `op (Operator)` – The operator to translate along the spin chain.
- `size (int, optional)` – The size of the support of the resulting operator. If not provided, defaults to the value of `Operator.L`.
- `start (int, optional)` – The site for the first operator in the sum.

Saved Operators

dynamite.operators.load_from_file(`filename`)
Load the operator in file `filename` and return the corresponding object.

Parameters `filename (str)` – The path of the file to load.

Returns The operator as a dynamite object.

Return type dynamite.operators.Load
dynamite.operators.from_bytes(`data`)
Load operator from a byte string generated with the `Operator.serialize()` method.

Parameters `data (bytes)` – The byte string containing the serialized object.

Returns The operator.
class dynamite.operators.Operator

A class representing a quantum operator.

This class generally won’t be directly instantiated by the user, but is returned by the other functions in this module.

copy()

Return a copy of the operator. Copy will not have its PETSc matrix already built, even if the operator being copied does.

Return type Operator

Parameters

- state (dynamite.states.State) – The initial state.
- t (float) – The time \(t\) for which to evolve the state (can be negative or complex).
- **kwargs – Any further keyword arguments are passed to the underlying call to dynamite.computations.evolve(). See that documentation for a detailed description of possible arguments.

Returns The result vector \(\Psi_f\).

Return type dynamite.states.State

eigsolve(**kwargs)

Find eigenvalues (and eigenvectors if requested) of the Hamiltonian. This class method is a wrapper on dynamite.computations.eigsolve(). Any keyword arguments are passed to that function; see its documentation for details.

By default, finds one (or possibly a few) eigenvalues with the smallest real values (i.e. the ground state).

Note: The spin chain length \(L\) must be set before calling eigsolve.

Returns Either a 1D numpy array of eigenvalues, or a pair containing that array and a list of the corresponding eigenvectors.

Return type numpy.array or tuple(numpy.array, list(dynamite.states.State))

max_spin_idx

Read-only property giving the largest spin index on which this operator has support.

L

Property representing the length of the spin chain. If \(L\) hasn’t been set, defaults to the size of support of the operator (from site 0).
get_length()  
Returns the length of the spin chain for this operator. It is defined by the property `Operator.L()` if it has been set by the user. Otherwise, the number of sites on which the operator has support is returned by default.

dim  
Read-only attribute returning the dimensions of the matrix.

nnz  
The number of nonzero elements per row of the sparse matrix.

msc_size  
The number of elements in the MSC representation of the matrix.

density  
The density of the sparse matrix—that is, the number of non-zero elements per row divided by the length of a row.

Note: This quantity is not always well defined when using a subspace, since it can vary by row. In that case, the returned quantity will be an upper bound.

shell  
Switch whether to use shell matrices or not. For a description of shell matrices and their benefits, see the documentation.

Note: Changing this value after the matrix has been built will invoke a call to `Operator.destroy_mat()`.

left_subspace  
Get the default left subspace for this operator. This is the subspace most recently added with `Operator.add_subspace()`, or `config.subspace` if `Operator.add_subspace()` has not been called.

right_subspace  
Get the default right subspace for this operator. This is the subspace most recently added with `Operator.add_subspace()`, or `config.subspace` if `Operator.add_subspace()` has not been called.

subspace  
Get the default subspace for this operator. This is the subspace most recently added with `Operator.add_subspace()`, or `config.subspace` if `Operator.add_subspace()` has not been called.

add_subspace (left, right=None)  
Add a pair of subspaces that this operator is compatible with.

Parameters
  
  • **left** (`dynamite.subspaces.Subspace`) – A subspace the operator can map to (or multiply from the left)
  
  • **right** (`dynamite.subspaces.Subspace`, *optional*) – A subspace the operator can map from (or multiply to the right). If omitted, the left subspace is reused for the right.

get_subspace_list()  
Return a list of the subspaces that have been registered for this operator.

string  
A text string that will be used to represent the object in printed expressions.
A LaTeX expression corresponding to the object. Can be set to any valid TeX.

Which kind of brackets to surround the expression with. Options are '()' , '[]' , or '', where the empty string means no brackets.

Return a string or tex representation of the object, surrounded by brackets if necessary. Useful for building larger expressions.

Parameters which (str) – Whether to return a normal string or tex. Options are 'string' or 'tex'.

Return a string containing an ASCII table of the coefficients and terms that make up this operator.

The table is generated directly from the MSC representation, so it is expanded and simplified to the same form no matter how the operator was built.

Call Operator.reduce_msc() first for a more compact table.

Return a clean LaTeX representation of the operator.

Serialize the operator’s MSC representation into a string of bytes. The byte string ONLY contains the MSC representation and the spin chain length. It does not save any other information, such as subspaces etc.

Returns The byte string containing the serialized object.

Return type bytes

Save the MSC representation of the operator to disk. Can be loaded again through Load.

Parameters filename (str) – The path to the file to save the operator in.

Get the PETSc matrix corresponding to this operator, building it if necessary.

Parameters

• subspaces (tuple(Subspace, Subspace), optional) – The subspace pair to get the matrix for. If the matrix is already built for this pair, it will be reused. If this option is omitted, the last subspace added with Operator.add_subspace() will be used, or the Full space by default.

• diag_entries (bool, optional) – Ensure that the sparse matrix has all diagonal elements filled, even if they are zero. Some PETSc functions fail if the diagonal elements do not exist. Currently a dummy argument; diagonal entries are always included.

Returns The PETSc matrix corresponding to the operator.

Return type petsc4py.PETSc.Mat
**build_mat***(subspaces=None, diag_entries=False)***

Build the PETSc matrix, destroying any matrix that has already been built, and store it internally. This function does not return the matrix—see **Operator.get_mat()** for that functionality. This function is rarely needed by the end user, since it is called automatically whenever the underlying matrix needs to be built or rebuilt.

**destroy_mat***(subspaces=None)***

Destroy the PETSc matrix, freeing the corresponding memory. If the PETSc matrix does not exist (has not been built or has already been destroyed), the function has no effect.

**Parameters**

**subspaces** *(tuple(Subspace, Subspace), optional)* — Destroy only the matrix for a particular pair of subspaces.

**create_states()**

Return a bra and ket compatible with this matrix.

**Returns**

The two states

**Return type**

tuple

**msc**

The (mask, sign, coefficient) representation of the operator. This representation is used internally by dynamite.

**reduce_msc()**

Combine and sort terms in the MSC representation, compressing it and preparing it for use in the backend.

**is_reduced**

Whether **Operators.reduce_msc()** has been called. Can also be set manually to avoid calling that function, if you are sure that the terms are sorted already.

**get_shifted_msc**(shift, wrap_idx=None)

Get the MSC representation of the operator, with all terms translated along the spin chain (away from zero) by shift spins.

**Parameters**

- **shift** *(int)* — Shift the whole operator along the spin chain by shift spins.
- **wrap** *(bool)* — The site at which to begin wrapping around to the beginning of the spin chain. e.g. takes a site index i to i % wrap_idx. If None, do not wrap.

**Returns**

A numpy array containing the representation.

**Return type**

numpy.ndarray

**to_numpy**(subspaces=None, sparse=True)

Get a SciPy sparse matrix or dense numpy array representing the operator.

**Parameters**

- **subspaces** *(tuple(Subspace, Subspace), optional)* — The subspaces for which to get the matrix. If this option is omitted, the last subspace added with **Operator.add_subspace()** will be used, or the Full space by default.
- **sparse** *(bool, optional)* — Whether to return a sparse matrix or a dense array.

**Returns**

The array

**Return type**

np.ndarray(dtype = np.complex128)

**spy**(subspaces=None, max_size=1024)

Use matplotlib to show the nonzero structure of the matrix.

**Parameters**
• **subspaces** ([tuple(Subspace, Subspace), optional]) – The pair of subspaces for which to plot the matrix. Defaults to the most recent added with the Operator.add_subspace method, or otherwise config.subspace.

• **max_size** ([int, optional]) – The maximum matrix dimension for which this function can be called. Calling it for too large a matrix will not be informative and probably run out of memory, so this is a small safeguard.

**dot** (*x*, *result=None*)
Compute the matrix-vector product \( \vec{y} = A\vec{x} \)

**Parameters**

• **x** ([dynamite.states.State]) – The input state x.

• **result** ([dynamite.states.State, optional]) – A state in which to store the result. If omitted, a new State object is created.

**Returns** The result

**Return type** [dynamite.states.State]

**scale** (*x*)
Scale an operator by a numerical value without making a copy. This is more efficient than just doing \( x*\)Operator.

**Parameters**

• **x** ([numeric type]) – The coefficient to scale by

1.4.2 **dynamite.states**

class [dynamite.states.State](L=None, subspace=None, state=None, seed=None)

**Bases:** object

Class representing a state.

**Parameters**

• **L** ([int]) – Spin chain length. Can be ommitted if config.L is set.

• **subspace** ([dynamite.subspace.Subspace, optional]) – The subspace on which the state should be defined. If not specified, defaults to config.subspace.

• **state** ([int or str, optional]) – An initial product state to set the state to. Also accepts 'random'. The state can also be initialized later with the set_product() and set_random() methods.

• **seed** ([int, optional]) – If the state argument is set to 'random', the seed for the random number generator. This argument is ignored otherwise.

**copy** (*result=None*)

**subspace**
The space on which the vector is defined.

See [dynamite.subspaces](#) for details.

**vec**
The PETSc vector containing the state data.

**petsc4py** Vec methods can be used through this interface—for example, to find the norm of a State s, one can do `state.vec.norm()`. The methods don’t seem to be documented anywhere, but are fairly transparent from looking at the petsc4py source code.

1.4. **Full API documentation** 13
**classmethod str_to_state(s, L)**
Convert a string to an integer whose bitwise representation is the spin configuration (0=↑, 1=↓) of a product state. The characters ‘D’ and ‘U’ represent down and up spins, like "DUDDU...UDU" (D=↓, U=↑).

**Note:** For the string representation, the leftmost character is spin index 0. For an integer representation, the rightmost (least significant) bit is!

**Parameters**
- **s** (*int or string*) – The state. If an integer is passed, the same integer is returned.
- **L** (*int*) – The length of the spin chain

**Returns** The state

**Return type** int

**set_product(s)**
Initialize to a product state. Can be specified either be an integer whose binary representation represents the spin configuration (0=↑, 1=↓) of a product state, or a string of the form "DUDDU...UDU" (D=↓, U=↑). If it is a string, the string’s length must equal L.

**Parameters**
- **s** (*int or str*) – A representation of the state.

**classmethod generate_time_seed()**

**set_random(seed=None, normalize=True)**
Initialized to a normalized random state.

**Note:** When running under MPI with multiple processes, the seed is incremented by the MPI rank, so that each process generates different random values.

**Parameters**
- **seed** (*int, optional*) – A seed for numpy’s PRNG that is used to build the random state. The user should pass the same value on every process.
- **normalize** (*bool*) – Whether to rescale the random state to have norm 1.

**to_numpy(to_all=False)**
Return a numpy representation of the state.

**Parameters**
- **to_all** (*bool*) – Whether to return the vector on all MPI ranks (True), or just rank 0 (False).

**dot(x)**

**norm(f='norm')**

**normalize(f='normalize')**

### 1.4.3 dynamite.subspaces

Classes that define the various subspaces on which operators can be defined.
The methods generally are just an interface to the backend, so that there is only one implementation of each of the functions.
class dynamite.subspaces.Subspace

Bases: object

Base subspace class.

check_L(value)

L

The spin chain length corresponding to this space.

get_dimension()

Get the dimension of the subspace.

dx_to_state(idx)

Maps an index to an integer that in binary corresponds to the spin configuration. Vectorized implementation allows passing a numpy array of indices as idx.

state_to_idx(state)

The inverse mapping of idx_to_state().

copy()

get_checksum()

Get a checksum of the state mapping for this subspace. This allows subspaces to be compared quickly.

get_cdata()

Returns an object containing the subspace data accessible by the backend C.

to_enum()

Convert the class types used in the Python frontend to the enum values used in the C backend.

class dynamite.subspaces.Full

Bases: dynamite.subspaces.Subspace

get_dimension()

Get the dimension of the subspace.

dx_to_state(idx)

Maps an index to an integer that in binary corresponds to the spin configuration. Vectorized implementation allows passing a numpy array of indices as idx.

state_to_idx(state)

The inverse mapping of idx_to_state().

get_cdata()

Returns an object containing the subspace data accessible by the C backend.

to_enum()

Convert the class types used in the Python frontend to the enum values used in the C backend.

class dynamite.subspaces.Parity(space)

Bases: dynamite.subspaces.Subspace

The subspaces of states in which the number of up spins is even or odd.

Parameters

space(int) – Either 0 or ‘even’ for the even subspace, or 1 or ‘odd’ for the other.

space

get_dimension()

Get the dimension of the subspace.

dx_to_state(idx)

Maps an index to an integer that in binary corresponds to the spin configuration. Vectorized implementation allows passing a numpy array of indices as idx.
state_to_idx(state)
The inverse mapping of idx_to_state().

get_cdata()
Returns an object containing the subspace data accessible by the C backend.

to_enum()
Convert the class types used in the Python frontend to the enum values used in the C backend.

class dynamite.subspaces.Auto(H, state, size_guess=None, sort=True)
Bases: dynamite.subspaces.Subspace
Automatically generate a mapping that takes advantage of any possible spin conservation law, by performing a
breadth-first search of the graph of possible states using the operator as an adjacency matrix. The subspace is
defined by providing a “start” state; the returned subspace will be whatever subspace contains that state.
Currently the actual computation of the ordering only can occur on process 0, limiting the scalability of this
subspace.

Parameters
• H (dynamite.operators.Operator) – The operator for which this custom subspace
  will be defined.
• state (int or string) – An integer whose binary representation corresponds to the
  spin configuration of the “start” state mentioned above, or string representing the same. See
dynamite.states.State.str_to_state() for more information.
• size_guess (int) – A guess for the dimension of the subspace. By default, memory is
  allocated for the full space, and then trimmed off if not used.
• sort (bool) – Whether to reorder the mapping after computing it. In some cases this may
  cause a speedup.
check_L(value)
get_dimension()
Get the dimension of the subspace.
idx_to_state(idx)
Maps an index to an integer that in binary corresponds to the spin configuration. Vectorized implementa-
tion allows passing a numpy array of indices as idx.
state_to_idx(state)
The inverse mapping of idx_to_state().

get_cdata()
Returns an object containing the subspace data accessible by the C backend.

to_enum()
Convert the class types used in the Python frontend to the enum values used in the C backend.

1.4.4 dynamite.computations
dynamite.computations.evolve(H, state, t, result=None, **kwargs)
Evolve a quantum state according to the Schrodinger equation under the Hamiltonian H. The units are natural,
that is, the evolution is simply
\[ \Psi_t = e^{-iHt}\Psi_0 \]

Parameters
• **H** *(Operator)* – The Hamiltonian

• **state** *(dynamite.states.State)* – A dynamite State object containing the state to be evolved.

• **t** *(float)* – The time for which to evolve. Can be negative to evolve backwards in time.

• **result** *(dynamite.states.State, optional)* – Where to store the result state. If not given, a new vector is created in which to store the result. If evolving repeatedly many times, it is a good idea to pass a result vector to avoid repeatedly allocating a lot of memory. Will be overwritten.

• **tol** *(float, optional)* – The tolerance for the evolution. Error estimation is difficult for Krylov exponentiation; this merely requests that the error be somewhat close to tol. There is no guarantee that it will actually be smaller.

• **algo** *(string, optional)* – Allowed options: ‘krylov’ or ‘expokit’. Which SLEPc algorithm to use to compute the matrix exponential.

• **ncv** *(int, optional)* – The Krylov subspace size to use. Increasing subspace size can increase performance by reducing the number of iterations necessary, but also linearly increases memory usage and the number of matrix multiplies performed. Optimizing this parameter can significantly affect performance.

Returns The result state

Return type *dynamite.states.State*

dynamite.computations.eigsolve *(H, getvecs=False, nev=1, which='smallest', target=None, tol=None, subspace=None)*

Solve for a subset of the eigenpairs of the Hamiltonian.

By default, solves for the eigenvalue with the smallest (most negative) real part, e.g. the ground state. Which eigenvalues are sought and how many can be adjusted with the options.

**Note:** Krylov algorithms have difficulty with degenerate or very nearly degenerate eigenvalues. Degenerate eigenvalues may be missed, and near-degenerate eigenstates may be inaccurate.

**Note:** Do not try to use this function to solve for the whole spectrum! It’s very efficient at finding a few eigenvalues, but no faster than other routines for finding all of them. In the future an efficient solver for the whole spectrum may be included with dynamite.

**Parameters**

• **getvecs** *(Bool)* – Whether to return eigenvectors as well as eigenvalues.

• **nev** *(int)* – The number of eigenvalues sought. The algorithm may return more eigenvalues than nev if more happen to converge.

• **which** *(str)* – Which eigenvalues to seek. Options are:
  - "smallest", to find the eigenvalues with smallest real part (i.e. most negative)
  - "largest", to find the eigenvalues with largest real part (i.e. most positive)
  - "exterior", to find eigenvalues largest in absolute magnitude
  - "target", to find eigenvalues closest to the given target

If **target** is set, **which** can be omitted and will automatically be set to "target".
• **target** *(float)* – Using the shift-invert method, the eigensolver can seek the eigenvalues with real part closest to some target value. This requires a linear solve and so will be slower than solving for exterior eigenvalues. PETSc must be configured with a parallel linear solver (e.g. --download-mumps option in configure) to use this option in parallel.

• **tol** *(float)* – The tolerance for the computation.

• **subspace** *(dynamite.subspaces.Subspace, optional)* – The subspace on which to solve for eigenvalues. If not given, defaults to the most recent subspace set with Operator.add_subspace, or config.subspace if no subspaces have been added.

Returns Either a 1D numpy array of eigenvalues, or a pair containing that array and a list of the corresponding eigenvectors.

Return type numpy.array or tuple(numpy.array, list(dynamite.states.State))

dynamite.computations.reduced_density_matrix*(state, keep)*

Compute the reduced density matrix of a state vector by tracing out some set of spins. The spins to be kept (not traced out) are specified in the keep array.

The density matrix is returned on process 0, the function returns a 1x1 matrix containing the value -1 on all other processes.

Parameters

• **state** *(dynamite.states.State)* – A dynamite State object.

• **keep** *(array-like)* – A list of spin indices to keep. Must be sorted. Note that the returned matrix will have dimension 2^len(keep), so too long a list will generate a huge matrix.

Returns The density matrix

Return type numpy.ndarray[np.complex128]

dynamite.computations.entanglement_entropy*(state, keep)*

Compute the entanglement of a state across some cut on the spin chain. To be precise, this is the bipartite entropy of entanglement.

Currently, this quantity is computed entirely on process 0. As a result, the function returns -1 on all other processes.

Parameters

• **state** *(dynamite.states.State)* – A dynamite State object.

• **keep** *(array-like)* – A list of spin indices to keep. See reduced_density_matrix() for details.

Returns The entanglement entropy

Return type float

dynamite.computations.dm_entanglement_entropy*(dm)*

Compute the Von Neumann entropy of a density matrix.

Parameters **dm** *(np.array)* – A density matrix

Returns The Von Neumann entropy

Return type float

dynamite.computations.renyi_entropy*(state, keep, alpha, method='eigsolve')*

Compute the Renyi entropy of the density matrix that results from tracing out some spins. The Renyi entropy is

\[ H_\alpha = \frac{1}{1 - \alpha} \log \text{Tr} \left[ \rho^\alpha \right] \]
Arbitrary non-negative values of $\alpha$ are allowed; in the special cases of $\alpha \in \{0, 1\}$ the function is computed in the limit.

Currently, this quantity is computed entirely on process 0. As a result, the function returns $-1$ on all other processes.

**Parameters**

- **state** (*dynamite.states.State*) – A dynamite State object.
- **keep** (*array-like*) – A list of spin indices to keep. See `reduced_density_matrix()` for details.
- **alpha** (*float, int, or str*) – The value of $\alpha$ from the definition of Renyi entropy.
- **method** (*str, optional*) – Whether to compute the Renyi entropy by solving for eigenvalues, or computing a matrix power and doing a trace. One or the other may be faster depending on the specific problem. Options: `eigsolve` or `matrix_power`.

**Returns** The Renyi entropy

**Return type** float

```python
dynamite.computations.dm_renyi_entropy(dm, alpha, method='eigsolve')
```

Compute the Renyi entropy of a density matrix. See `renyi_entropy()` for details.

**Parameters**

- **dm** (*np.array*) – A density matrix
- **alpha** (*int, float, or str*) – The value of alpha in the definition of Renyi entropy.
- **method** (*str*) – Whether to compute the Renyi entropy by solving for eigenvalues, or computing a matrix power and doing a trace. One or the other may be faster depending on the specific problem. Options: `eigsolve` or `matrix_power`.

**Returns** The Renyi entropy

**Return type** float

```python
dynamite.computations.get_tstep(ncv, nrm, tol=1e-07)
```

Compute the length of a sub-step in a Expokit matrix exponential solve.

```python
dynamite.computations.estimate_compute_time(t, ncv, nrm, tol=1e-07)
```

Estimate compute time in units of matrix multiplies, for an expokit exponential solve.

### 1.4.5 `dynamite.tools`

Various tools useful for writing and analyzing dynamite programs.

```python
dynamite.tools.mpi_print(*args, rank=0, **kwargs)
```

Print from only a single MPI rank, default rank 0.

Aside from the extra “rank” keywork argument, call signature is the same as Python 3’s `print()` function.

```python
dynamite.tools.get_version()
```

Gets the version information for dynamite, and the PETSc and SLEPc libraries it’s built on.

**Returns** A dictionary with the keys ‘PETSc’, ‘SLEPc’, and ‘dynamite’, each of which contains version information for the respective library.

**Return type** dict

```python
dynamite.tools.get_version_str()
```

Get a string with the version information for PETSc, SLEPc, and dynamite.


**dynamite Documentation, Release 0.1.0 [beta]**

Returns  The version string
Return type  str

dynamite.tools.track_memory()
Begin tracking memory usage for a later call to get_max_memory_usage().

dynamite.tools.get_max_memory_usage(which='all')
Get the maximum memory usage up to this point. Only updated whenever objects are destroyed (e.g. with dynamite.operators.Operator.destroy_mat())

**Note:** track_memory() must be called before this function is called, and the option '-malloc' must be supplied to PETSc at runtime if which == 'petsc'.

Parameters **which** (str) – 'all' to return all memory usage for the process, 'petsc' to return only memory allocated by PETSc.

Returns  The max memory usage in bytes
Return type  float

dynamite.tools.get_cur_memory_usage(which='all')
Get the current memory usage (resident set size) in bytes.

Parameters **type** (str) – 'all' to return all memory usage for the process, 'petsc' to return only memory allocated by PETSc.

Returns  The max memory usage in bytes
Return type  float

### 1.4.6 dynamite.extras

dynamite.extras.commutator(o1, o2)
The commutator \([O_1, O_2]\).

Returns  The commutator
Return type  dynamite.operators.Operator

dynamite.extras.majorana(idx)
A function generating an operator that represents a Majorana fermion as a boundary in a spin chain.
The boundary is at index \(b\_idx = \text{floor}(idx/2) + 1\). The operator consists of the tensor product of \(\sigma_z\) operators up to spin \(b\_idx - 1\), and then on spin \(b\_idx\) a \(\sigma_x\) operator if \(idx\) is even or a \(\sigma_y\) operator if \(idx\) is odd.

Parameters **idx** (int) – The index of the Majorana

Returns  The Majorana of index \(idx\)
Return type  dynamite.operators.Operator

### 1.4.7 dynamite.config

The dynamite._Config class is instantiated by dynamite as dynamite.config. This is the object on which one should call these functions. For example:
from dynamite import config

config.initialize(['-mfn_ncv','20'])
config.L = 24

class dynamite._Config
    Package-wide configuration of dynamite.
    
    initialize(slepc_args=None, version_check=True, gpu=False)
        Initialize PETSc/SLEPc with various arguments (which would be passed on the command line for a C program).
        
        Only the first call to this function has any effect. It is automatically called when using much of the PETSc/SLEPc functionality (including importing petsc4py.PETSc or slepc4py.SLEPc), so it must be called early (probably right after importing dynamite).
        
        Parameters

        • slepc_args(list of str) – The arguments to SLEPc initialization.

        • version_check(bool) – Whether process 0 should check for a new dynamite version on initialization. Can be set to false if the check is unnecessary or causes problems.

        • gpu(bool) – Whether to run all computations using a GPU instead of the CPU.

    L
        A global spin chain length that will be applied to all matrices and states, unless they are explicitly set to a different size. Is not retroactive—will not set the size for any objects that have already been created.

    shell
        Whether to use standard PETSc matrices (False, default), or shell matrices (True).

    subspace
        The subspace to use for all operators and states. Can also be set for individual operators and states—see dynamite.operators.Operator.subspace for details.

    gpu
        Whether to run the computations on a GPU. This property is read-only. To use GPUs, initialize() must be called with gpu=True.

This package was created by Greg Meyer in Prof. Norman Yao’s lab at UC Berkeley.
Python Module Index

d
dynamite.computations, 16
dynamite.extras, 20
dynamite.states, 13
dynamite.subspaces, 14
dynamite.tools, 19
Symbols

_Config (class in dynamite), 21

A
add_subspace() (dynamite.operators.Operator method), 10
Auto (class in dynamite.subspaces), 16

B
brackets (dynamite.operators.Operator attribute), 11
build_mat() (dynamite.operators.Operator method), 11

C
check_L() (dynamite.subspaces.Auto method), 16
check_L() (dynamite.subspaces.Subspace method), 15
commutator() (in module dynamite.extras), 15
copy() (dynamite.operators.Operator method), 9
copy() (dynamite.states.State method), 13
copy() (dynamite.subspaces.Subspace method), 15
create_states() (dynamite.operators.Operator method), 12

density (dynamite.operators.Operator attribute), 10
destroy_mat() (dynamite.operators.Operator method), 12
dim (dynamite.operators.Operator attribute), 10
dm_entanglement_entropy() (in module dynamite.computations), 18
dm_renyi_entropy() (in module dynamite.computations), 19
dot() (dynamite.operators.Operator method), 13
dot() (dynamite.states.State method), 14
dynamite.computations (module), 14
dynamite.extras (module), 20
dynamite.states (module), 13
dynamite.subspaces (module), 15
dynamite.tools (module), 19

E
eigsolve() (dynamite.operators.Operator method), 9
eigsolve() (in module dynamite.computations), 17
entanglement_entropy() (in module dynamite.computations), 18
estimate_compute_time() (in module dynamite.computations), 19
evolve() (dynamite.operators.Operator method), 9
evolve() (in module dynamite.computations), 16

F
from_bytes() (in module dynamite.operators), 8
Full (class in dynamite.subspaces), 15

generate_time_seed() (dynamite.states.State class method), 14
get_cdata() (dynamite.subspaces.Auto method), 16
get_cdata() (dynamite.subspaces.Full method), 15
get_cdata() (dynamite.subspaces.Parity method), 16
get_cdata() (dynamite.subspaces.Subspace method), 15
get_checksum() (dynamite.subspaces.Subspace method), 15
get_cur_memory_usage() (in module dynamite.tools), 20
get_dimension() (dynamite.subspaces.Auto method), 16
get_dimension() (dynamite.subspaces.Full method), 15
get_dimension() (dynamite.subspaces.Parity method), 15
get_dimension() (dynamite.subspaces.Subspace method), 15
get_latex() (dynamite.operators.Operator method), 11
get_length() (dynamite.operators.Operator method), 9
get_mat() (dynamite.operators.Operator method), 11
get_max_memory_usage() (in module dynamite.tools), 20
get_shifted_msc() (dynamite.operators.Operator method), 12
get_subspace_list() (dynamite.operators.Operator method), 19
get_tstep() (in module dynamite.computations), 10
get_version() (in module dynamite.computations), 19
get_version_str() (in module dynamite.tools), 19
gpu (dynamite._Config attribute), 21

identity() (in module dynamite.operators), 7
idx_to_state() (dynamite.subspaces.Auto method), 16
idx_to_state() (dynamite.subspaces.Full method), 15
idx_to_state() (dynamite.subspaces.Parity method), 15
idx_to_state() (dynamite.subspaces.Subspace method), 15
index_product() (in module dynamite.operators), 8
index_sum() (in module dynamite.operators), 8
initialize() (dynamite._Config method), 21
is_reduced (dynamite.operators.Operator attribute), 12

L
L (dynamite._Config attribute), 21
L (dynamite.operators.Operator attribute), 9
L (dynamite.subspaces.Subspace attribute), 15
left_subspace (dynamite.operators.Operator attribute), 10
load_from_file() (in module dynamite.operators), 8

M
majorana() (in module dynamite.extras), 20
max_spin_idx (dynamite.operators.Operator attribute), 9
mpi_print() (in module dynamite.tools), 19
msc (dynamite.operators.Operator attribute), 12
msc_size (dynamite.operators.Operator attribute), 10

N
nnz (dynamite.operators.Operator attribute), 10
norm() (dynamite.states.State method), 14
normalize() (dynamite.states.State method), 14

O
op_product() (in module dynamite.operators), 7
op_sum() (in module dynamite.operators), 7
Operator (class in dynamite.operators), 9

P
Parity (class in dynamite.subspaces), 15

R
reduce_msc() (dynamite.operators.Operator method), 12
reduced_density_matrix() (in module dynamite.computations), 18
renyi_entropy() (in module dynamite.computations), 18
right_subspace (dynamite.operators.Operator attribute), 10

S
save() (dynamite.operators.Operator method), 11
scale() (dynamite.operators.Operator method), 13
serialize() (dynamite.operators.Operator method), 11
set_product() (dynamite.states.State method), 14
set_random() (dynamite.states.State method), 14
shell (dynamite._Config attribute), 21
shell (dynamite.operators.Operator attribute), 10
sigma_minus() (in module dynamite.operators), 7
sigma_plus() (in module dynamite.operators), 7
sigmax() (in module dynamite.operators), 7
sigmay() (in module dynamite.operators), 7
sigmaz() (in module dynamite.operators), 7
space (dynamite.subspaces.Parity attribute), 15
spy() (dynamite.operators.Operator method), 12
State (class in dynamite.states), 13
state_to_idx() (dynamite.subspaces.Auto method), 16
state_to_idx() (dynamite.subspaces.Full method), 15
state_to_idx() (dynamite.subspaces.Parity method), 15
state_to_idx() (dynamite.subspaces.Subspace method), 15
str_to_state() (dynamite.states.State class method), 13
string (dynamite.operators.Operator attribute), 10
Subspace (class in dynamite.subspaces), 14
subspace (dynamite._Config attribute), 21
subspace (dynamite.operators.Operator attribute), 10
subspace (dynamite.states.State attribute), 13

T
table() (dynamite.operators.Operator method), 11
tex (dynamite.operators.Operator attribute), 10
to_enum() (dynamite.subspaces.Auto method), 16
to_enum() (dynamite.subspaces.Full method), 15
to_enum() (dynamite.subspaces.Parity method), 16
to_enum() (dynamite.subspaces.Subspace method), 15
to_numpy() (dynamite.operators.Operator method), 12

to_numpy() (dynamite.states.State method), 14

track_memory() (in module dynamite.tools), 20

V

vec (dynamite.states.State attribute), 13

W

with_brackets() (dynamite.operators.Operator method), 11

Z

zero() (in module dynamite.operators), 7