spinbox

Release 0.1

Jordan M. R. Fox

Aug 13, 2024

CONTENTS:

1 spinbox **1**
1.1 spinbox package 1

2 Indices and tables **21**

Python Module Index **23**

Index **25**

CHAPTER

ONE

SPINBOX

1.1 spinbox package

1.1.1 Submodules

1.1.2 spinbox.core module

class spinbox.core.**CoulombCoupling**(*n_particles*, *file=None*)

Bases: *Coupling*

container class for couplings V^coul (i,j) for i, j = 0 .. n_particles - 1

random(*scale*, *seed=0*)

validate()

class spinbox.core.**Coupling**(*n_particles: int*, *shape: tuple[int]*, *file=None*)

Bases: `object`

Base class for coupling arrays.

Set and get are defined like numpy.ndarray objects.

The simplest example would be something like $g_{\alpha i}$ for $\alpha = x, y, z$ and $i = 0 \ldots A-1$.

```
A = 2
g = Coupling(n_particles=A, shape=(3,A))    #initialize to zeros
g[0,0] = 1.0  # set an entry by hand
```

copy()

read(*filename*)

class spinbox.core.**ExactPropagator**(*n_particles*, *isospin=True*)

Bases: `object`

The "exact" propagator.

$$\exp\left(-\sum_n g_n \hat{v}_n\right)$$

where g_n is the entire scalar factor (e.g. $\frac{\delta\tau}{2} A^{\sigma}_{i\alpha j\beta}$, note the phase convention) and $\hat{v}_n$ is the 2- or 3-body interaction operator.

Note, this calculation must be done in the complete many-body basis; it cannot be restricted to product states.

We use a Pade approximant for the matrix exponential. The LS term can be represented using a linear approximation or the factorization procedure described in Stefano's thesis.

Returns
The exact propagator.

Return type
HilbertOperator

force_coulomb(*coupling:* CoulombCoupling, *i: int*, *j: int*) → *HilbertOperator*

force_coulomb_onebody(*coupling: complex*, *i: int*) → *HilbertOperator*

just the one-body part of the expanded coulomb propagator for use along with auxiliary field propagators

force_sigma(*coupling:* SigmaCoupling, *i: int*, *j: int*) → *HilbertOperator*

force_sigma_3b(*g:* ThreeBodyCoupling, *i: int*, *j: int*, *k: int*) → *HilbertOperator*

force_sigmatau(*coupling:* SigmaTauCoupling, *i: int*, *j: int*) → *HilbertOperator*

force_tau(*coupling:* TauCoupling, *i: int*, *j: int*) → *HilbertOperator*

propagator_combined(*dt*, *potential*, *sigma=False*, *sigmatau=False*, *tau=False*, *coulomb=False*, *spinorbit=False*, *sigma_3b=False*)

propagator_spinorbit_linear(*coupling:* SpinOrbitCoupling, *i: int*) → *HilbertOperator*

propagator_spinorbit_onebody(*g:* SpinOrbitCoupling, *i: int*) → *HilbertOperator*

propagator_spinorbit_twobody(*g:* SpinOrbitCoupling, *i: int*, *j: int*) → *HilbertOperator*

class spinbox.core.**HilbertOperator**(*n_particles: int*, *isospin=True*)

Bases: `object`

An operator in the "Hilbert basis.

apply_onebody_operator(*particle_index: int*, *spin_matrix: ndarray*, *isospin_matrix: ndarray = None*)

Applies a one-body / single-particle operator to the `HilbertOperator`. This accounts for the spin-isospin kronecker product, if isospin is used.

$$O' = \sigma_{\alpha i} \tau_{\beta i} O$$

Parameters

- **particle_index** (*int*) – Index of particle to apply onebody operator to, starting from 0.
- **spin_matrix** (*np.ndarray*) – The spin part of the operator, a 2x2 matrix
- **isospin_matrix** (*numpy.ndarray, optional*) – The isospin part of the operator, a 2x2 matrix, defaults to None

Returns
A copy of the `HilbertOperator` with the one-body operator applied.

Return type
HilbertOperator

apply_sigma(*particle_index: int*, *dimension: int*) → *HilbertOperator*

Applies a one-body sigma spin operator.

Parameters

- **particle_index** (*int*) – Index of particle, staring from 0.
- **dimension** (*int*) – Dimension of sigma operator: 0, 1, 2 = x, y, z

Returns
The resulting `HilbertOperator`.

Return type
HilbertOperator

apply_tau(*particle_index: int*, *dimension: int*)

Applies a one-body tau isospin operator.

Parameters

- **particle_index** (*int*) – Index of particle, staring from 0.
- **dimension** (*int*) – Dimension of tau operator: 0, 1, 2 = x, y, z

Returns
The resulting `HilbertOperator`.

Return type
HilbertOperator

copy() → *HilbertOperator*

Copies the `HilbertOperator`.

Returns
a new instance of `HilbertOperator` with all the same properties as self.

Return type
HilbertOperator

dagger() → *HilbertOperator*

Hermitian conjugate.

Returns
The Hermitian conjugate of the original `HilbertOperator`.

Return type
HilbertOperator

exp() → *HilbertOperator*

Computes the exponential by Pade approximant.

Returns
Exponentiated operator.

Return type
HilbertOperator

multiply_operator(*other:* HilbertOperator) → *HilbertOperator*

Multiply two `HilbertOperator` instances together to get a new one.

Parameters
other (`HilbertOperator`) – The other `HilbertOperator`

Returns
The product of the two.

Return type
HilbertOperator

multiply_state(*other:* HilbertState) → *HilbertState*

Apply the operator to a `HilbertState` ket.

Parameters
other (`HilbertState`) – The state, ketwise.

Returns
The new state, ketwise.

Return type
HilbertState

scale(*other: complex*) → *HilbertOperator*

Scalar multiplication.

Parameters
other (*complex*) – A scalar.

Returns
The resulting scaled operator.

Return type
HilbertOperator

zero() → *HilbertOperator*

Multiplies by zero.

Returns
A copy of the `HilbertOperator` with all zero coefficients.

Return type
HilbertOperator

class spinbox.core.**HilbertPropagatorHS**(*n_particles: int*, *dt: float*, *isospin=True*, *include_prefactors=True*)

Bases: *Propagator*

The two-body propagator applied by Hubbard-Stratonovich

$$\exp\left[-\frac{\delta\tau}{2}\sum_{\alpha i\beta j} A_{\alpha i\beta j}\hat{o}_{\alpha i}\hat{o}_{\beta j}\right]$$

factors_coulomb(*coupling:* Coupling, *aux: list*) → list[*HilbertOperator*]

Creates factors of the Coulomb propagator.

$$\exp\left[-\frac{\delta\tau}{2}\frac{v_{ij}}{4}(1+\tau_{iz}+\tau_{jz}+\tau_{iz}\tau_{jz})\right]$$

The result is a list of (noncommuting) terms so they may be shuffled.

Parameters

- **coupling** (`Coupling`) – force coupling array (e.g. $v_C(r_{ij})$)

- **aux** (*list*) – values of auxiliary field, length equal to the number of pairs

Returns
The list of propagator terms

Return type
list[*HilbertOperator*]

factors_sigma(*coupling:* Coupling, *aux: list*) → list[*HilbertOperator*]

Creates factors of the $A^{\sigma}_{\alpha i \beta j}\sigma_{i\alpha}\sigma_{j\beta}$ propagator. The result is a list of (noncommuting) terms so they may be shuffled.

Parameters

- **coupling** (Coupling) – force coupling array (e.g. $A^{\sigma}_{\alpha i \beta j}$)
- **aux** (*list*) – values of auxiliary field, length equal to the number of pairs*3*3

Returns
The list of propagator terms

Return type
list[*HilbertOperator*]

factors_sigmatau(*coupling:* Coupling, *aux: list*) → list[*HilbertOperator*]

Creates factors of the $A^{\sigma\tau}_{\alpha i \beta j}\sigma_{i\alpha}\sigma_{j\beta}\tau_{i\gamma}\tau_{j\gamma}$ propagator. The result is a list of (noncommuting) terms so they may be shuffled.

Parameters

- **coupling** (Coupling) – force coupling array (e.g. $A^{\sigma\tau}_{\alpha i \beta j}$)
- **aux** (*list*) – values of auxiliary field, length equal to the number of pairs*3*3*3

Returns
The list of propagator terms

Return type
list[*HilbertOperator*]

factors_spinorbit(*coupling:* Coupling, *aux: list*) → list[*HilbertOperator*]

Creates factors of the spin-orbit propagator.

$$\exp\left[-\frac{\delta\tau}{2}v_{LS}(r_{ij})\mathbf{L}\cdot\mathbf{S}\right]$$

The result is a list of (noncommuting) terms so they may be shuffled.

Parameters

- **coupling** (Coupling) – force coupling array (e.g. $g^{\mathrm{LS}}_{\alpha i}$)
- **aux** (*list*) – values of auxiliary field, length equal to the number of pairs

Returns
The list of propagator terms

Return type
list[*HilbertOperator*]

factors_tau(*coupling:* Coupling, *aux: list*)

Creates factors of the $\tau_{i\gamma}\tau_{j\gamma}$ propagator. The result is a list of (noncommuting) terms so they may be shuffled.

Parameters

- **coupling** (Coupling) – force coupling array (e.g. A^{τ}_{ij})
- **aux** (*list*) – values of auxiliary field, length equal to the number of pairs*3

Returns
The list of propagator terms

Return type
list[*HilbertOperator*]

onebody(*z: complex*, *operator:* HilbertOperator) → *HilbertOperator*

A one-body propagator

$$\exp\left[-z\hat{o}\right]$$

Parameters

- **z** (*complex*) – scalar
- **operator** (HilbertOperator) – one-body operator

Returns
The one-body propagator

Return type
HilbertOperator

twobody_sample(*z: complex*, *x: float*, *operator_i:* HilbertOperator, *operator_j:* HilbertOperator) → *HilbertOperator*

A sample of the two-body propagator in the integrand of the Hubbard-Stratonovich transform.

$$\exp(z)\exp(x\sqrt{-z}\hat{\sigma}_{i\alpha})\exp(x\sqrt{-z}\hat{\sigma}_{j\beta})$$

Parameters

- **z** (*complex*) – scalar
- **x** (*float*) – auxiliary field value
- **operator_i** (HilbertOperator) – operator on particle i
- **operator_j** (HilbertOperator) – operator on particle j

Returns
One sample of the two-body propagator

Return type
HilbertOperator

class spinbox.core.**HilbertPropagatorRBM**(*n_particles*, *dt: float*, *isospin=True*, *include_prefactors=True*)

Bases: *Propagator*

exp(- i z op_i op_j)

factors_coulomb(*coupling:* CoulombCoupling, *aux: list*)

factors_sigma(*coupling:* SigmaCoupling, *aux: list*)

factors_sigma_3b(*coupling:* ThreeBodyCoupling, *aux: list*)

factors_sigmatau(*coupling:* SigmaTauCoupling, *aux: list*)

factors_spinorbit(*coupling:* SpinOrbitCoupling, *aux: list*)

factors_tau(*coupling:* TauCoupling, *aux: list*)

onebody(*z: complex*, *operator:* HilbertOperator)

exp (- z opi)

threebody_sample(*z: float*, *h_list: list*, *onebody_matrix_i*, *onebody_matrix_j*, *onebody_matrix_k*)

three body RBM sample written for one combined 3-body RBM kernel function

threebody_sample_partial(*z: float*, *h_list: list*, *operator_i:* HilbertOperator, *operator_j:* HilbertOperator, *operator_k:* HilbertOperator)

three body propagator sample using three 2-body RBMs

twobody_sample(*z: float*, *h: int*, *operator_i:* HilbertOperator, *operator_j:* HilbertOperator)

class spinbox.core.**HilbertState**(*n_particles: int*, *coefficients=None*, *ketwise=True*, *isospin=True*)

Bases: `object`

A spin state in the "Hilbert" basis, a linear combination of tensor product states.

States must be defined with a number of particles. If `isospin` is False, then the one-body basis is only spin up/down. If True, then it is (spin up/down x isospin up/down). `ketwise` detemines if it is a bra or a ket.

attach_coordinates(*coordinates: ndarray*)

Adds a new `.coordinates` attribute to the `HilbertState`

Parameters
coordinates (*np.ndarray*) – A Numpy array with shape `(n_particles , 3)` (e.g. x, y, z)

copy()

Copies the `HilbertState`.

Returns
a new instance of `HilbertState` with all the same properties.

Return type
HilbertState

dagger() → *HilbertState*

Hermitian conjugate.

Returns
The dual `HilbertState`

Return type
HilbertState

entropy() → complex

Von Neumann entropy, a measure of entanglement.

Returns
VN entropy of the `HilbertState`

Return type
complex

generate_basis_states() → list

Makes a list of corresponding basis vectors.

Returns
A list of tensor product states that span the Hilbert space.

Return type
list

inner(*other:* HilbertState) → complex

Inner product of two HilbertState instances. Orientations must be correct.

Parameters
other (HilbertState) – The ket of the inner product.

Returns
inner product of self (bra) with other (ket)

Return type
complex

multiply_operator(*other:* HilbertOperator) → *HilbertState*

Multiplies a (bra) HilbertState on a HilbertOperator.

Parameters
other (HilbertOperator) – The operator.

Returns
< self| O(other)

Return type
HilbertState

nearby_product_state(*seed: int = None*, *maxiter=100*)

Finds a ProductState that has a large overlap with the HilbertState.

Parameters

- **seed** (*int, optional*) – RNG seed, defaults to None
- **maxiter** (*int, optional*) – maximum iterations to do in optimization, defaults to 100

Returns
a tuple: (fitted ProductState, optimization result)

Return type
(*ProductState*, scipy.OptimizeResult)

nearest_product_state(*seeds: list[int]*, *maxiter=100*)

Does self.nearby_product_state for a list of seeds and returns the result maximizing overlap

Parameters

- **seed** (*int, optional*) – RNG seed, defaults to None
- **maxiter** (*int, optional*) – maximum iterations to do in optimization, defaults to 100

Returns
fitted ProductState

Return type
ProductState

outer(*other:* HilbertState) → *HilbertOperator*

Outer product of two HilbertState instances, producting a HilbertOperator instance. Orientations must be correct.

Parameters
other (HilbertState) – bra part of the outer product

Returns
Outer product of self (ket) with other (bra)

Return type
HilbertOperator

randomize(*seed: int = None*) → *HilbertState*

Randomize coefficients.

Parameters
seed (*int, optional*) – RNG seed, defaults to None

Returns
A copy of the `HilbertState` with random complex coefficients, normalized.

Return type
HilbertState

scale(*other: complex*) → *HilbertState*

Scalar multiple of a `HilbertState`.

Parameters
other (*complex*) – Scalar number to multiply by.

Returns
other * self

Return type
HilbertState

zero() → *HilbertState*

Set all coefficients to zero.

Returns
A copy of `HilbertState` with all coefficients set to zero.

Return type
HilbertState

class spinbox.core.**Integrator**(*potential: NuclearPotential*, *propagator*, *isospin=True*)

Bases: `object`

bracket(*bra*, *ket*, *aux_fields*)

exact(*bra*, *ket*)

run(*bra*, *ket*)

setup(*n_samples*, *seed=0*, *mix=True*, *flip_aux=False*, *sigma=False*, *sigmatau=False*, *tau=False*, *coulomb=False*, *spinorbit=False*, *sigma_3b=False*, *parallel=True*, *n_processes=None*)

class spinbox.core.**NuclearPotential**(*n_particles*)

Bases: `object`

container class for Argonne-style NN potential + NNN

read_coulomb(*filename*)

read_sigma(*filename*)

read_sigma_3b(*filename*)

read_sigmatau(*filename*)

read_spinorbit(*filename*)

read_tau(*filename*)

class spinbox.core.**ProductOperator**(*n_particles: int*, *isospin=True*)

Bases: `object`

An operator that is a tensor product of one-body operators.

As with `ProductState` instances, `ProductOperator` instances cannot be added or subtracted.

apply_onebody_operator(*particle_index: int*, *spin_matrix: ndarray*, *isospin_matrix: ndarray = None*) → *ProductOperator*

Applies a one-body / single-particle operator to the `ProductOperator`. This accounts for the spin-isospin kronecker product, if isospin is used.

$$O' = \sigma_{\alpha i} \tau_{\beta i} O$$

Parameters

- **particle_index** (*int*) – Index of particle to apply onebody operator to, starting from 0.
- **spin_matrix** (*np.ndarray*) – The spin part of the operator, a 2x2 matrix
- **isospin_matrix** (*numpy.ndarray, optional*) – The isospin part of the operator, a 2x2 matrix, defaults to None

Returns
A copy of the `ProductOperator` with the one-body operator applied.

Return type
ProductOperator

apply_sigma(*particle_index: int*, *dimension: int*) → *ProductOperator*

Applies a one-body sigma spin operator.

Parameters

- **particle_index** (*int*) – Index of particle, staring from 0.
- **dimension** (*int*) – Dimension of sigma operator: 0, 1, 2 = x, y, z

Returns
The resulting `ProductOperator`.

Return type
ProductOperator

apply_tau(*particle_index: int*, *dimension: int*) → *ProductOperator*

Applies a one-body tau isospin operator.

Parameters

- **particle_index** (*int*) – Index of particle, staring from 0.
- **dimension** (*int*) – Dimension of tau operator: 0, 1, 2 = x, y, z

Returns
The resulting `ProductOperator`.

Return type
ProductOperator

copy() → *ProductOperator*

Copies the ProductOperator.

Returns
a new instance of ProductOperator with all the same properties as self.

Return type
ProductOperator

dagger()

Hermitian conjugate.

Returns
The dual ProductOperator

Return type
ProductOperator

multiply_operator(*other:* ProductOperator) → *ProductOperator*

Multiply two ProductOperator instances together to get a new one.

Parameters
other (ProductOperator) – The other ProductOperator

Returns
The product of the two.

Return type
ProductOperator

multiply_state(*other:* ProductState) → *ProductState*

Apply the operator to a ProductState ket.

Parameters
other (ProductState) – The state, ketwise.

Returns
The new state, ketwise.

Return type
ProductState

scale_all(*b*) → *ProductOperator*

Scales an A-body operator by b by multiplying each one-body matrix by the Ath root of b.

Parameters
b (*complex*) – scalar

Returns
The scaled state

Return type
ProductOperator

scale_one(*particle_index: int*, *b: complex*) → *ProductOperator*

Multiplies a single particle operator matrix by a number.

Parameters

- **particle_index** (*int*) – Index of particle, starting from 0.
- **b** (*complex*) – Scalar

Returns
A copy of the `ProductOperator` with the one-body matrix scaled.

Return type
ProductOperator

`to_list()` → list[ndarray]

Returns
A list of one-body operator matrices

Return type
list[numpy.ndarray]

`to_manybody_basis()`

Projects to the many-body basis.

Returns
The Kronecker product of the `ProductOperator`.

Return type
HilbertOperator

`zero()` → *ProductOperator*

Set all coefficients to zero.

Returns
A copy of `ProductOperator` with all coefficients set to zero.

Return type
ProductOperator

`class spinbox.core.ProductPropagatorHS(n_particles: int, dt: float, isospin=True, include_prefactors=True)`

Bases: *Propagator*

the propagator exp(- i z op_i op_j)

`factors_coulomb(coupling: CoulombCoupling, aux: list)`

`factors_sigma(coupling: SigmaCoupling, aux: list)`

`factors_sigmatau(coupling: SigmaTauCoupling, aux: list)`

`factors_spinorbit(coupling: SpinOrbitCoupling, aux: list)`

`factors_tau(coupling: TauCoupling, aux: list)`

`onebody(z: complex, i: int, onebody_matrix: ndarray)`

exp (- z opi)

`twobody_sample(z: complex, x: float, i: int, j: int, onebody_matrix_i: ndarray, onebody_matrix_j: ndarray)`

exp (x * sqrt(-z) opi) exp (x * sqrt(-z) opj)

`class spinbox.core.ProductPropagatorRBM(n_particles, dt, isospin=True, include_prefactors=True)`

Bases: *Propagator*

exp(- i z op_i op_j) seed determines mixing

`factors_coulomb(coupling: CoulombCoupling, aux: list)`

factors_sigma(*coupling:* SigmaCoupling, *aux: list*)

factors_sigma_3b(*coupling:* ThreeBodyCoupling, *aux: list*)

factors_sigmatau(*coupling:* SigmaTauCoupling, *aux: list*)

factors_spinorbit(*coupling:* SpinOrbitCoupling, *aux: list*)

factors_tau(*coupling:* TauCoupling, *aux: list*)

onebody(*z: complex*, *i: int*, *onebody_matrix: ndarray*)

exp (- i z opi)

threebody_sample(*z: float*, *h_list: list*, *i: int*, *j: int*, *k: int*, *onebody_matrix_i*, *onebody_matrix_j*, *onebody_matrix_k*)

three body RBM sample written for one combined 3-body RBM kernel function

twobody_sample(*z: complex*, *h: int*, *i: int*, *j: int*, *onebody_matrix_i*, *onebody_matrix_j*)

class spinbox.core.**ProductState**(*n_particles: int*, *coefficients=None*, *ketwise=True*, *isospin=True*)

Bases: `object`

A spin state in the "Product" basis, a single tensor product of one-body vectors.

States must be defined with a number of particles. If `isospin` is False, then the one-body basis is only spin up/down. If True, then it is (spin up/down x isospin up/down). `ketwise` detemines if it is a bra or a ket.

Tensor product states do not form a proper vector space (e.g. the sum of two is not guaranteed to be a tensor product) so methods with `ProductState` are restricted. Namely operations + and - do not exist.

The coefficients of the `ProductState` are kept in the one-body form and can be projected to the Hilbert basis using the `to_manybody_basis` method.

attach_coordinates(*coordinates: ndarray*)

Adds a new `.coordinates` attribute to the `ProductState`

Parameters

coordinates (*np.ndarray*) – A Numpy array with shape `(n_particles , 3)` (e.g. x, y, z)

copy()

Copies the `ProductState`.

Returns

a new instance of `ProductState` with all the same properties.

Return type

ProductState

dagger() → *ProductState*

Hermitian conjugate.

Returns

The dual `ProductState`

Return type

ProductState

generate_basis_states() → list[*ProductState*]

Makes a list of corresponding basis vectors.

Returns
A list of tensor product states that span the Hilbert space.

Return type
list[*ProductState*]

`inner`(*other:* ProductState) → complex

Inner product of two ProductState instances. Orientations must be correct.

Parameters
other (ProductState) – The ket of the inner product.

Returns
inner product of self (bra) with other (ket)

Return type
complex

`normalize`() → *ProductState*

Normalize so that the inner product of the state with itself is 1.

Returns
The normalized state.

Return type
ProductState

`outer`(*other:* ProductState) → *ProductState*

Outer product of two ProductState instances, producting a ProductOperator instance. Orientations must be correct.

Parameters
other (ProductState) – bra part of the outer product

Returns
Outer product of self (ket) with other (bra)

Return type
ProductOperator

`randomize`(*seed: int = None*) → *ProductState*

Randomize coefficients.

Parameters
seed (*int, optional*) – RNG seed, defaults to None

Returns
A copy of the ProductState with random complex coefficients, normalized.

Return type
ProductState

`scale_all`(*b: complex*) → *ProductState*

Scales an A-body state by b by multiplying each one-body vector by the Ath root of b.

Parameters
b (*complex*) – scalar

Returns
The scaled state

Return type
ProductState

scale_one(*particle_index: int*, *b: complex*) → *ProductState*

Multiplies a single particle vector by a number.

Parameters

- **particle_index** (*int*) – Index of particle, starting from 0.
- **b** (*complex*) – Scalar

Returns
A copy of the ProductState with the one particle scaled.

Return type
ProductState

to_list() → list

Returns
A list of one-body vectors

Return type
list[numpy.ndarray]

to_manybody_basis() → *HilbertState*

Projects to the many-body basis.

Returns
The Kronecker product of the ProductState.

Return type
HilbertState

zero() → *ProductState*

Set all coefficients to zero.

Returns
A copy of ProductState with all coefficients set to zero.

Return type
ProductState

class spinbox.core.**Propagator**(*n_particles*, *dt: float*, *isospin=True*, *include_prefactors=True*)

Bases: object

class spinbox.core.**SigmaCoupling**(*n_particles*, *file=None*)

Bases: *Coupling*

The coupling matrix $A^{\sigma}_{\alpha i \beta j}$

for i, j = 0 .. n_particles - 1 and a, b = 0, 1, 2 (x, y, z)

random(*scale*, *seed=0*)

validate()

class spinbox.core.**SigmaTauCoupling**(*n_particles*, *file=None*)

Bases: *Coupling*

container class for couplings A ^ sigma tau (a,i,b,j) for i, j = 0 .. n_particles - 1 and a, b = 0, 1, 2 (x, y, z)

Note that there are no dimensional indices for tau because the tau factor is a dot product, and thus the couplings are the same over dimensions.

random(*scale*, *seed=0*)

validate()

class spinbox.core.**SpinOrbitCoupling**(*n_particles*, *file=None*)

Bases: *Coupling*

container class for couplings g_LS (a,i) for i = 0 .. n_particles - 1 and a = 0, 1, 2 (x, y, z)

random(*scale*, *seed=0*)

validate()

class spinbox.core.**TauCoupling**(*n_particles*, *file=None*)

Bases: *Coupling*

container class for couplings A^tau (i,j) for i, j = 0 .. n_particles - 1

random(*scale*, *seed=0*)

validate()

class spinbox.core.**ThreeBodyCoupling**(*n_particles*, *file=None*)

Bases: *Coupling*

container class for couplings A(a,i,b,j,c,k) for i, j, k = 0 .. n_particles - 1 and a = 0, 1, 2 (x, y, z)

random(*scale*, *seed=0*)

validate()

spinbox.core.**carctanh**(*x*)

Complex incerse hyp. tangent

spinbox.core.**ccos**(*x*)

Complex cosine

spinbox.core.**ccosh**(*x*)

Complex hyp. cosine

spinbox.core.**cexp**(*x*)

Complex exponential

spinbox.core.**csin**(*x*)

Complex sine

spinbox.core.**csinh**(*x*)

Complex hyp. sine

spinbox.core.**csqrt**(*x*)

Complex square root

spinbox.core.**ctanh**(*x*)

Complex hyp. tangent

spinbox.core.**interaction_indices**(*n: int*, *m=2*) → list

returns a list of all possible m-plets of n objects (labelled 0 to n-1) default: m=2, giving all possible pairs for m=1, returns a range(0, n-1) :param n: number of items :type n: int :param m: size of tuplet, defaults to 2 :type m: int, optional :return: list of possible m-plets of n items :rtype: list

spinbox.core.**pauli**(*arg*) → ndarray

Pauli matrix x, y, z, or a list of all three

Parameters
arg (*int or str*) – 0 or 'x' for Pauli x, 1 or 'y' for Pauli y, 2 or 'z' for Pauli z, 'list' for a list of x, y ,z

Raises
ValueError – option not found

Returns
Pauli matrix or list

Return type
np.ndarray

spinbox.core.**read_from_file**(*filename: str*, *complex=False*, *shape=None*, *order='F'*) → ndarray

Read numbers from a text file

Parameters

- **filename** (*str*) – input file name
- **complex** (*bool, optional*) – complex entries, defaults to False
- **shape** (*tuple, optional*) – shape of output array, defaults to None
- **order** (*str, optional*) – 'F' for columns first, otherwise use 'C', defaults to 'F'

Returns
Numpy array

Return type
numpy.ndarray

spinbox.core.**repeated_kronecker_product**(*matrices: list*) → ndarray

returns the tensor/kronecker product of a list of arrays :param matrices: list of matrix factors :type matrices: list :return: Kronecker product of input list "rtype: np.ndarray

1.1.3 spinbox.extras module

spinbox.extras.**chistogram**(*X*, *filename*, *title*, *bins='fd'*, *range=None*)

Complex histogram

Parameters

- **X** (*iterable*) – Set of complex numbers
- **filename** (*str*) – filename of plot, including suffix (.pdf)
- **title** (*str*) – Plot title
- **bins** (*str, optional*) – binning algorithm (see matplotlib.pyplot.hist), defaults to 'fd' (Freedman-Diaconis)
- **range** (*tuple, optional*) – fixed range to plot, defaults to None

spinbox.extras.**pmat**(*x*, *heatmap=False*, *lims=None*, *print_zeros=False*)

Print or plot a complex-valued matrix

Parameters

- **x** (*numpy.ndarray*) – matrix to be plotted

- **heatmap** (*bool, optional*) – True if plotting a heatmap, defaults to False
- **lims** (*tuple, optional*) – if heatmap, limits for colorbar, defaults to None
- **print_zeros** (*bool, optional*) – True if printing a part if it is all zeros, defaults to False

spinbox.extras.**sigma_tau_matrices_product**(*n_particles*)

Pauli sigma and tau matrices in product basis

Parameters
n_particles (*int*) – number of particles

Returns
(sigma matrices, tau matrices)

Return type
tuple of lists

usage: sigma, tau = sigma_tau_matrices_product(n_particles) sigma[dimension_index]

note that I am not using the ProductOperator class here. This is done for memory efficiency. In the case of Hilbert space calculations, it makes sense to compute the operator matrices beforehand and store them. In the tensor-product basis, this would result in most of our memory being taken up by identity matrices.

spinbox.extras.**sigma_tau_operators_hilbert**(*n_particles*)

Pauli sigma and tau operators in Hilbert space

Parameters
n_particles (*int*) – number of particles

Returns
(sigma operators, tau operators)

Return type
tuple of lists of lists

usage: sigma, tau = sigma_tau_operators_hilbert(n_particles) sigma[particle_index][dimension_index]

spinbox.extras.**spinor2**(*state='up'*, *ketwise=True*, *seed=None*)

Convenience function for making 2-dimensional spin state vectors

Parameters

- **state** (*str, optional*) – can be one of [‘up’, ‘down’, ‘random’, ‘max’], defaults to ‘up’.
- **ketwise** (*bool, optional*) – True for column vector, False for row vector, defaults to True
- **seed** (*int, optional*) – rng seed, defaults to None

Returns
your vector

Return type
numpy.ndarray

spinbox.extras.**spinor4**(*state='up'*, *ketwise=True*, *seed=None*)

Convenience function for making 4-dimensional spin-isospin state vectors

Parameters

- **state** (*str, optional*) – can be one of [‘up’, ‘down’, ‘random’, ‘max’], defaults to ‘up’.
- **ketwise** (*bool, optional*) – True for column vector, False for row vector, defaults to True
- **seed** (*int, optional*) – rng seed, defaults to None

Returns
your vector

Return type
numpy.ndarray

1.1.4 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

spinbox, 19
spinbox.core, 1
spinbox.extras, 17

INDEX

A

apply_onebody_operator() (*spinbox.core.HilbertOperator method*), 2
apply_onebody_operator() (*spinbox.core.ProductOperator method*), 10
apply_sigma() (*spinbox.core.HilbertOperator method*), 2
apply_sigma() (*spinbox.core.ProductOperator method*), 10
apply_tau() (*spinbox.core.HilbertOperator method*), 3
apply_tau() (*spinbox.core.ProductOperator method*), 10
attach_coordinates() (*spinbox.core.HilbertState method*), 7
attach_coordinates() (*spinbox.core.ProductState method*), 13

B

bracket() (*spinbox.core.Integrator method*), 9

C

carctanh() (*in module spinbox.core*), 16
ccos() (*in module spinbox.core*), 16
ccosh() (*in module spinbox.core*), 16
cexp() (*in module spinbox.core*), 16
chistogram() (*in module spinbox.extras*), 17
copy() (*spinbox.core.Coupling method*), 1
copy() (*spinbox.core.HilbertOperator method*), 3
copy() (*spinbox.core.HilbertState method*), 7
copy() (*spinbox.core.ProductOperator method*), 10
copy() (*spinbox.core.ProductState method*), 13
CoulombCoupling (*class in spinbox.core*), 1
Coupling (*class in spinbox.core*), 1
csin() (*in module spinbox.core*), 16
csinh() (*in module spinbox.core*), 16
csqrt() (*in module spinbox.core*), 16
ctanh() (*in module spinbox.core*), 16

D

dagger() (*spinbox.core.HilbertOperator method*), 3
dagger() (*spinbox.core.HilbertState method*), 7
dagger() (*spinbox.core.ProductOperator method*), 11
dagger() (*spinbox.core.ProductState method*), 13

E

entropy() (*spinbox.core.HilbertState method*), 7
exact() (*spinbox.core.Integrator method*), 9
ExactPropagator (*class in spinbox.core*), 1
exp() (*spinbox.core.HilbertOperator method*), 3

F

factors_coulomb() (*spinbox.core.HilbertPropagatorHS method*), 4
factors_coulomb() (*spinbox.core.HilbertPropagatorRBM method*), 6
factors_coulomb() (*spinbox.core.ProductPropagatorHS method*), 12
factors_coulomb() (*spinbox.core.ProductPropagatorRBM method*), 12
factors_sigma() (*spinbox.core.HilbertPropagatorHS method*), 5
factors_sigma() (*spinbox.core.HilbertPropagatorRBM method*), 6
factors_sigma() (*spinbox.core.ProductPropagatorHS method*), 12
factors_sigma() (*spinbox.core.ProductPropagatorRBM method*), 12
factors_sigma_3b() (*spinbox.core.HilbertPropagatorRBM method*), 6
factors_sigma_3b() (*spinbox.core.ProductPropagatorRBM method*), 13
factors_sigmatau() (*spinbox.core.HilbertPropagatorHS method*), 5
factors_sigmatau() (*spinbox.core.HilbertPropagatorRBM method*),

6
factors_sigmatau() (spinbox.core.ProductPropagatorHS method), 12
factors_sigmatau() (spinbox.core.ProductPropagatorRBM method), 13
factors_spinorbit() (spinbox.core.HilbertPropagatorHS method), 5
factors_spinorbit() (spinbox.core.HilbertPropagatorRBM method), 6
factors_spinorbit() (spinbox.core.ProductPropagatorHS method), 12
factors_spinorbit() (spinbox.core.ProductPropagatorRBM method), 13
factors_tau() (spinbox.core.HilbertPropagatorHS method), 5
factors_tau() (spinbox.core.HilbertPropagatorRBM method), 6
factors_tau() (spinbox.core.ProductPropagatorHS method), 12
factors_tau() (spinbox.core.ProductPropagatorRBM method), 13
force_coulomb() (spinbox.core.ExactPropagator method), 2
force_coulomb_onebody() (spinbox.core.ExactPropagator method), 2
force_sigma() (spinbox.core.ExactPropagator method), 2
force_sigma_3b() (spinbox.core.ExactPropagator method), 2
force_sigmatau() (spinbox.core.ExactPropagator method), 2
force_tau() (spinbox.core.ExactPropagator method), 2

G

generate_basis_states() (spinbox.core.HilbertState method), 7
generate_basis_states() (spinbox.core.ProductState method), 13

H

HilbertOperator (class in spinbox.core), 2
HilbertPropagatorHS (class in spinbox.core), 4
HilbertPropagatorRBM (class in spinbox.core), 6
HilbertState (class in spinbox.core), 7

I

inner() (spinbox.core.HilbertState method), 8
inner() (spinbox.core.ProductState method), 14
Integrator (class in spinbox.core), 9
interaction_indices() (in module spinbox.core), 16

M

module
    spinbox, 19
    spinbox.core, 1
    spinbox.extras, 17
multiply_operator() (spinbox.core.HilbertOperator method), 3
multiply_operator() (spinbox.core.HilbertState method), 8
multiply_operator() (spinbox.core.ProductOperator method), 11
multiply_state() (spinbox.core.HilbertOperator method), 4
multiply_state() (spinbox.core.ProductOperator method), 11

N

nearby_product_state() (spinbox.core.HilbertState method), 8
nearest_product_state() (spinbox.core.HilbertState method), 8
normalize() (spinbox.core.ProductState method), 14
NuclearPotential (class in spinbox.core), 9

O

onebody() (spinbox.core.HilbertPropagatorHS method), 6
onebody() (spinbox.core.HilbertPropagatorRBM method), 7
onebody() (spinbox.core.ProductPropagatorHS method), 12
onebody() (spinbox.core.ProductPropagatorRBM method), 13
outer() (spinbox.core.HilbertState method), 8
outer() (spinbox.core.ProductState method), 14

P

pauli() (in module spinbox.core), 16
pmat() (in module spinbox.extras), 17
ProductOperator (class in spinbox.core), 10
ProductPropagatorHS (class in spinbox.core), 12
ProductPropagatorRBM (class in spinbox.core), 12
ProductState (class in spinbox.core), 13
Propagator (class in spinbox.core), 15
propagator_combined() (spinbox.core.ExactPropagator method), 2
propagator_spinorbit_linear() (spinbox.core.ExactPropagator method), 2
propagator_spinorbit_onebody() (spinbox.core.ExactPropagator method), 2

propagator_spinorbit_twobody() (*spinbox.core.ExactPropagator method*), 2

R

random() (*spinbox.core.CoulombCoupling method*), 1
random() (*spinbox.core.SigmaCoupling method*), 15
random() (*spinbox.core.SigmaTauCoupling method*), 15
random() (*spinbox.core.SpinOrbitCoupling method*), 16
random() (*spinbox.core.TauCoupling method*), 16
random() (*spinbox.core.ThreeBodyCoupling method*), 16
randomize() (*spinbox.core.HilbertState method*), 9
randomize() (*spinbox.core.ProductState method*), 14
read() (*spinbox.core.Coupling method*), 1
read_coulomb() (*spinbox.core.NuclearPotential method*), 9
read_from_file() (*in module spinbox.core*), 17
read_sigma() (*spinbox.core.NuclearPotential method*), 9
read_sigma_3b() (*spinbox.core.NuclearPotential method*), 9
read_sigmatau() (*spinbox.core.NuclearPotential method*), 9
read_spinorbit() (*spinbox.core.NuclearPotential method*), 10
read_tau() (*spinbox.core.NuclearPotential method*), 10
repeated_kronecker_product() (*in module spinbox.core*), 17
run() (*spinbox.core.Integrator method*), 9

S

scale() (*spinbox.core.HilbertOperator method*), 4
scale() (*spinbox.core.HilbertState method*), 9
scale_all() (*spinbox.core.ProductOperator method*), 11
scale_all() (*spinbox.core.ProductState method*), 14
scale_one() (*spinbox.core.ProductOperator method*), 11
scale_one() (*spinbox.core.ProductState method*), 14
setup() (*spinbox.core.Integrator method*), 9
sigma_tau_matrices_product() (*in module spinbox.extras*), 18
sigma_tau_operators_hilbert() (*in module spinbox.extras*), 18
SigmaCoupling (*class in spinbox.core*), 15
SigmaTauCoupling (*class in spinbox.core*), 15
spinbox
    module, 19
spinbox.core
    module, 1
spinbox.extras
    module, 17
spinor2() (*in module spinbox.extras*), 18
spinor4() (*in module spinbox.extras*), 18
SpinOrbitCoupling (*class in spinbox.core*), 16

T

TauCoupling (*class in spinbox.core*), 16
threebody_sample() (*spinbox.core.HilbertPropagatorRBM method*), 7
threebody_sample() (*spinbox.core.ProductPropagatorRBM method*), 13
threebody_sample_partial() (*spinbox.core.HilbertPropagatorRBM method*), 7
ThreeBodyCoupling (*class in spinbox.core*), 16
to_list() (*spinbox.core.ProductOperator method*), 12
to_list() (*spinbox.core.ProductState method*), 15
to_manybody_basis() (*spinbox.core.ProductOperator method*), 12
to_manybody_basis() (*spinbox.core.ProductState method*), 15
twobody_sample() (*spinbox.core.HilbertPropagatorHS method*), 6
twobody_sample() (*spinbox.core.HilbertPropagatorRBM method*), 7
twobody_sample() (*spinbox.core.ProductPropagatorHS method*), 12
twobody_sample() (*spinbox.core.ProductPropagatorRBM method*), 13

V

validate() (*spinbox.core.CoulombCoupling method*), 1
validate() (*spinbox.core.SigmaCoupling method*), 15
validate() (*spinbox.core.SigmaTauCoupling method*), 16
validate() (*spinbox.core.SpinOrbitCoupling method*), 16
validate() (*spinbox.core.TauCoupling method*), 16
validate() (*spinbox.core.ThreeBodyCoupling method*), 16

Z

zero() (*spinbox.core.HilbertOperator method*), 4
zero() (*spinbox.core.HilbertState method*), 9
zero() (*spinbox.core.ProductOperator method*), 12
zero() (*spinbox.core.ProductState method*), 15