GElib

Group equivariant tensor Library, v0.0

Last update: August 2021

Risi Kondor
Department of Computer Science, Statistics,
and the Computational and Applied Math Initiative
The University of Chicago

Erik H Thiede
Center for Computational Mathematics
Flatiron Institute

Contents

Overview

GElib is a C++/CUDA library implementing various operations related to the representation theory of SO(3) and certain other groups. The primary purpose of GElib is to serve as a backend for building group equivariant neural networks that can take advantage of GPU architectures. The library is built on top of the cnine tensor library, https://github.com/risi-kondor/cnine.

GElib is under continuous development. Some features described in this document may not be fully implemented yet. GElib is authored by Risi Kondor and Erik H Thiede and released under the Mozilla Public License, version 2.0, https://www.mozilla.org/en-US/MPL/2.0/.

Installation

GElib is mostly a header library, and does not need to be installed in a specific location on your system. However, if the library is to be used with GPU functionality, certain CUDA object files do have to be separately compiled. This is done by running `make all` in the `cuda` directory.

GElib is an extension of the cnine library, which must be separately pulled from https://github.com/risi-kondor/cnine. Please refer to the cnine documentation for the description of the basic scalar and tensor classes used in GElib. The only other dependencies of GElib are:

- An appropriate `C++11` compiler together with the STL standard template library.
- CUDA and CUBLAS if the library is to be used with GPU functionality.

Compilation options

Before attempting to compile the CUDA objects or compile your own code against GElib, please copy `options_template.txt` to a file named `options.txt`, and set the following compilation parameters defined within.

Variable	Default value	Description
`CC`	`clang`	Name of C++ compiler.
`CNINE_ROOT`	`$(ROOTDIR)/../cnine/`	Root directory of the `cnine` installation.
`WITH_CUDA`	`t`	If this variable is defined, the library will link to CUDA. If not defined, GPU functionality is disabled.
`CUDA_DIR`	`/usr/local/cuda`	Root directory of the CUDA installation on your system.
`NVCC`	`nvcc`	Name of CUDA compiler.
`NVCCFLAGS`	omitted	Various flags to be passed to the NVCC compiler.
`WITH_CUBLAS`	`t`	If this variable is defined, the library will link to CUBLAS for low level linear algebra functionality on the GPU.

Usage

To call GElib from your own code C++ code you must do the following:

- `#include` the relevant header files in your soure files.
- `#include` the file `include/GElib_base.cpp` in your top level source file (the one that contains your `main` function).
- Define a `GElibSession` object before calling any cnine or cnine objects or functions.
- Link in the appropriate CUDA object files from the `cuda` directory, as required.

Reference

☜	This function produces a temporary that is assignable and therefore may be used as an lvalue.

`NBU`	Bundle dimension. If bundling is off, should be set to -1.
`DEVICE`	`device` object specifying the device on which a given variable is initialized or moved to. Currently the two possible values are `deviceid::CPU` and `deviceid::GPU0` or 0 and 1.

expr. template	shorthand	description
`Conjugate<OBJ>`	`conj(x)`	The conjugate of `x`.
`Transpose<OBJ>`	`transp(x)`	The transpose of `x`.
`Hermitian<OBJ>`	`herm(x)`	The Hermitian conjugate (conjugate transpose) of `x`.

SO(3) classes

SO3element

An `SO3element` object represents a rotation $R \in \mathrm{SO}(3)$.

CONSTRUCTORS

`SO3element(double phi, double theta, double psi)`
: The rotation parametrized by Euler angles (ϕ, θ, ψ).

`SO3element(fill::uniform)`
: A rotation chosen at random from (uniformly w.r.t. Haar measure on SO(3)).

SO3type

An `SO3type` object defines the type $\boldsymbol{\tau} = (\tau_0, \tau_1, \ldots, \tau_L)$ of an SO(3)–vector.

Derived from: `vector<int>`

N_1

CONSTRUCTORS

`SO3type(int L)`
: A new type vector $\boldsymbol{\tau} = (\tau_0, \tau_1, \ldots, \tau_L)$, where each τ_ℓ is initialized to zero.

`SO3type(initializer_list<int> list)`
: Initialize $\boldsymbol{\tau}$ from `list`.

MEMBER ACCESS

`int getL() const`
`int maxl() const`
: Return L.

`int operator(int l) const`
: Return τ_ℓ.

RELATED FUNCTIONS

`SO3type SO3type::CGproduct(SO3type& tau1, SO3type& tau2)`
`SO3type SO3type::CGproductroduct(SO3type& tau1, SO3type& tau2, int maxL)`
: Return the type of $\mathbf{v}_1 \otimes \mathbf{v}_2$, when $\mathbf{v}_1$ is of type τ_1 and $\mathbf{v}_2$ is of type τ_2. In the second form of the function, the result is band limited to $\ell_{\max}$

SO3part

An `SO3part` object stores a single isotypic part $P = P^\ell$ of an SO(3)–vector.

Implemented by: `SO3partA` (derived from `cnine::CtensorA`)

CONSTRUCTORS

`SO3part(int l, int n, [NBU], [FILLTYPE], [DEVICE])`
A new `SO3part` object for the ℓ'th isotypic consisting of n fragments. `FILLTYPE` can be `cnine::fill::raw` (default), `cnine::fill::zero` or `cnine::fill::gaussian`.

`SO3part(int l, int n, [NBU], std::function<complex<float>(const int i, const int m)> fn)`
A new `SO3part` object for the ℓ'th isotypic consisting of n fragments. The entries of P are computed by the function `fn`.

STATIC CONSTRUCTORS

`SO3part SO3part::zero(int l, int n, [NBU], [DEVICE])`
`SO3part SO3part::ones(int l, int n, [NBU], [DEVICE])`
`SO3part SO3part::gaussian(int l, int n, [NBU], [DEVICE])`
A new `SO3part` with the given fill pattern.

VARIANT CONSTRUCTORS

`SO3part(const SO3part& P, const device& dev)`
Create a copy of `P` on device `dev`.

`SO3part(const SO3part& P, const cnine::view)`
Create a view of `P`.

MEMBER ACCESS

`int getl() const`
Return the irrep index ℓ.

`int getn() const`
Return n, the number of irreducible fragments.

`int get_nbu() const`
Return the bundle dimension. If bundling is off, returns -1.

`int get_dev() const`
Return the device that this `SO3part` is stored on.

`Cscalar get(int i, int m) const`
`SO3part& set(int i, int m, const Cscalar& x)`
Return $[P_i]_m$ or set $[P_i]_m = x$ (note that $-\ell \le m \le \ell$).

`complex<float> get_value(int i, int m) const`
`SO3part& set_value(int i, int m, complex<float> x)`
Return the value of $[P_i]_m$ or set $[P_i]_m = x$ (note that $-\ell \le m \le \ell$).

`operator()(const int i, const int m)` ☜
Return a temporary for $[P_i]_m$ (note that $-\ell \le m \le \ell$).

`fragment(int i) const` ☜
Return a temporary for the i'th fragment of P.

`fragments(int i, int n=1) const` ☜
Return a temporary for the chunk of P consisting of fragments $(i, \ldots, i+n-1)$.

`SO3part apply(std::function<complex<float>(complex<float> x)> fn)`
`SO3part apply(std::function<complex<float>(const int i, const int m, complex<float> x)> fn)`
Return the `SO3part` derived by applying the function `fn` to each element of P.

ARITHMETIC

`c*P`
`c*P` `(complex<float>,SO3part) -> SO3part`
`P*c` `(cscalar,SO3part) -> SO3part`
`P*c` `(SO3part,complex<float>) -> SO3part`
`P/c` `(SO3part,cscalar) -> SO3part`
`P/c`
Multiply/divide P by the scalar c.

`P1+P2` `(SO3part,SO3part) -> SO3part`
`P1-P2` `(SO3part,SO3part) -> SO3part`
Compute $P_1 + P_2$ resp. $P_1 - P_2$. The parts must have the same ℓ and the same number of fragments.

`M*P` `(Ctensor,SO3part) -> SO3part`
Multiply P by the matrix M.

IN-PLACE OPERATIONS

`+=P2` `(SO3part,SO3part) -> SO3part`
`-=P2` `(SO3part,SO3part) -> SO3part`

Increment/decrement P by P_2.

MEMBER FUNCTIONS

`SO3part rotate(const SO3element& R) const`

Rotate P by the rotation $R \in \mathrm{SO}(3)$.

CG-PRODUCTS

`CGproduct(P1,P2,l)` `(SO3part,SO3part,int) -> SO3part`

Return the ℓ'th component of the Clebsch–Gordan product, $[P_1 \otimes_{cg} P_2]_\ell$.

OTHER FUNCTIONS

`norm2(P)` `SO3part -> Cscalar`

The squared Frobenius norm $\|P\|^2_{\mathrm{Frob}}$.

`inp(P1,P2)` `(SO3part,SO3part) -> Cscalar`

The inner product $\langle P_1, P_2 \rangle$.

I/O

`string str(const indent="") const`

Print `P` to string with the optional indentation `indent`.

SO3vec

An `SO3vec` object stores an SO(3)–covariant vector $\mathbf{v}$ in "Fourier form", i.e., as a collection of isotypic parts $(P^0, P^1, \ldots, P^L)$.

CONSTRUCTORS

`SO3vec(const SO3part& P0, ..., const SO3part& PL)`
A new SO(3)–vector consisting of parts $P^0, P^1, \ldots, P^L$.

`SO3vec(const SO3type& tau, [NBU], [FILLTYPE], [FORMAT], [DEVICE])`
A new `SO3vec` object of type τ. `FILLTYPE` may be `fill::raw` (default), `fill::zero` or `fill::gaussian`. `FORMAT` may be `SO3vec_format::parts` (default), or `SO3vec_format::compact`.

STATIC CONSTRUCTORS

`SO3vec SO3vec::zero(const SO3type& tau, [NBU], [FORMAT], [DEVICE])`
`SO3vec SO3vec::ones(const SO3type& tau, [NBU], [FORMAT], [DEVICE])`
`SO3vec SO3vec::gaussian(const SO3type& tau, [NBU], [FORMAT], [DEVICE])`
A new `SO3vec` of type τ initialized with the given fill pattern.

VARIANT CONSTRUCTORS

`SO3part(const SO3vec& v, const device& dev)`
Create a copy of `v` on device `dev`.

`SO3part(const SO3vec& v, const SO3vec_format& format)`
Create a copy of `v` with format `format`.

`SO3part(const SO3vec& v, const cnine::view)`
Create a view of `v`.

MEMBER ACCESS

`int getL() const`
Return L, the highest ℓ index.

`SO3type get_tau() const`
Return τ, the type of $\mathbf{v}$.

`int get_nbu() const`
Return the bundle dimension. If bundling is off, returns -1.

`int get_dev() const`
Return the device that this `SO3vec` is stored on.

`int get_format() const`
Return the device the storage format of this `SO3vec`.

`SO3part get_part(int l)`
`SO3vec& set_part(int, l, const SO3part& P)`
Get/set the ℓ'th isotypic part of `v`.

`SO3part get(int l)` ☜
Return a temporary to the ℓ'th isotypic part of $\mathbf{v}$.

ARITHMETIC

`c*v` `(cscalar,SO3vec) -> SO3vec`
`c*v` `(complex<float>,SO3vec) -> SO3vec`
`v*c` `(SO3vec,cscalar) -> SO3vec`
`v*c` `(SO3vec,complex<float>) -> SO3vec`
`v/c` `(SO3vec,cscalar) -> SO3vec`
`v/c` `(SO3vec,complex<float>) -> SO3vec`
Multiply/divide $\mathbf{v}$ by the complex scalar c.

`u+v` `(SO3vec,SO3vec) -> SO3vec`
`u-v` `(SO3vec,SO3vec) -> SO3vec`
Compute $\mathbf{u}+\mathbf{v}$ resp $\mathbf{u}-\mathbf{v}$. The two vectors $\mathbf{u}$ and $\mathbf{v}$ must be of the same type.

`W*v` `(CtensorPack,SO3vec) -> SO3vec`
Multiply $\mathbf{v}$ by the sequence of matrices stored in the `CtensorPack` `W`.

IN-PLACE OPERATIONS

`+=u` `(SO3vec,SO3vec) -> SO3vec`
`-=u` `(SO3vec,SO3vec) -> SO3vec`
Increment/decrement $\mathbf{v}$ by $\mathbf{u}$.

MEMBER FUNCTIONS

`SO3vec rotate(const SO3element& R) const`
Rotate $\mathbf{v}$ by the rotation $R \in \mathrm{SO}(3)$.

OTHER FUNCTIONS

`norm2(v)` `SO3vec -> Cscalar`

The squared Frobenius norm $\sum_\ell \| [\mathbf{v}]_\ell \|^2_{\text{Frob}}$.

`inp(u,v)` `(SO3vec,vec) -> Cscalar`

The inner product $\langle \mathbf{u}, \mathbf{v} \rangle$.

`inp(u,v,l)` `(SO3vec,SO3vec,l) -> Cscalar`

The inner product between the ℓ'th part of the inputs, $\langle [\mathbf{u}]_\ell, [\mathbf{v}]_\ell \rangle$.

CG-PRODUCTS

`CGproduct(u,v,[lmax])` `(SO3vec,SO3vec,int) -> SO3vec`

Compute the Clebsch–Gordan product $\mathbf{u} \otimes_{cg} \mathbf{v}$ up to $\ell = \ell_{\max}$.

I/O

`string str(const indent="") const`

Print `v` to string with the optional indentation `indent`.

SO3partArray

An `SO3partArray` $[\![P]\!]$ is an array of `SO3part` objects.

Implemented by: `SO3partArrayA` (derived from `cnine::CtensorArrayA`)

CONSTRUCTORS

`SO3partArray(const Gdims& adims, int l, int n, [NBU], [FILLTYPE], [DEVICE])`
Create an array of `SO3part` objects. The array dimensions are specified in `adims`. `FILLTYPE` can be `fill::raw` (default), `fill::zero` or `fill::gaussian`.

`SO3partArray(const Gdims& adims, const SO3part& P, [DEVICE])`
Create an `SO3partArray` in which each cell is a copy of `P`.

`SO3partArray(const Gdims& adims, int l, int n, [NBU], fill::view, float* arr, float* arrc, [DEVICE])`
Construct an `SO3partArray` that is a view of the data at `arr` (real part) and `arrc` (imaginary part).

STATIC CONSTRUCTORS

`SO3partArray SO3partArray::zero(const Gdims& adims, int l, int n, [NBU], [DEVICE])`
`SO3partArray SO3partArray::ones(const Gdims& adims, int l, int n, [NBU], [DEVICE])`
`SO3partArray SO3partArray::gaussian(const Gdims& adims, int l, int n, [NBU], [DEVICE])`
Create a new `SO3partArray` of array dimensions `adims` using the approriate fill pattern.

VARIANT CONSTRUCTORS

`SO3partArray(const SO3partArray& V, const device& dev)`
Create a copy of $[\![P]\!]$ on device `dev`.

RESHAPING

`SO3partArray shape(const Gdims& dims) const`
Return a copy of the `SO3partArray` reshaped to dimensions `dims`.

`SO3partArray& reshape(const Gdims& dims)`
Change the shape of the array to `dims`. The total number of cells must be preserved.

`SO3partArray as_shape(const Gdims& dims) const`
Return a temporary that reinterprets the shape of the array as `dims`.

MEMBER ACCESS

`int getl() const`
Return the irrep index ℓ.

`int getn() const`
Return τ_ℓ, the number of irreducible fragments in this part.

`int get_nbu() const`
Return the bundle dimension. If bundling is off, returns -1.

`int get_dev() const`
Return the device that this `SO3partArray` is stored on.

`Gdims get_adims()`
Return the array dimensions.

`SO3part get_cell(int i1,...,int ik) const`
Return the cell $[\![P]\!](i_1,\ldots,i_k)$.

`CtensorArray& set_cell(int i1,...,int ik, const SO3part& A)`
Set $[\![P]\!](i_1,\ldots,i_k)$=A.

`SO3part get_cell(Gindex& aix) const`
Return the cell $[\![P]\!](i_1,\ldots,i_k)$.

`CtensorArray& set_cell(Gindex& aix, const SO3part& A)`
Set $[\![P]\!](i_1,\ldots,i_k) = A$.

`SO3part cell(const Gindex& aix)` ☜
Return a temporary `SO3part` capturing cell `aix`.

ARRAY OPERATIONS

`SO3partArray broaden(int ix, int n) const`
Create a $d+1$ dimensional array by stacking n copies of $[\![P]\!]$ along dimension ix.

`SO3partArray reduce(int ix) const`
Create a $d-1$ dimensional array by summing out dimension `ix`.

`SO3part reduce() const`
Reduce $[\![P]\!]$ by summing all entries into a single `SO3part`.

`reshape(const Gdims& adims)`
Reinterpret $[\![P]\!]$ as an array of dimensions `adims`.

CELL-BY-CELL ARITHMETIC

`+=P2` `(SO3partArray,SO3partArray)->SO3partArray`
`-=P2` `(SO3partArray,SO3partArray)->SO3partArray`

Increment/decrement $[\![P]\!]$ by $[\![P_2]\!]$.

`P1+P2` `(SO3partArray,SO3partArray) -> SO3partArray`
`P1-P2` `(SO3partArray,SO3partArray) -> SO3partArray`

Compute $[\![P_1]\!]+[\![P_2]\!]$ resp. $[\![P_1]\!]-[\![P_2]\!]$.

`M*P` `(CtensorArray,SO3partArray) -> SO3partArray`

Multiply each cell of $[\![P]\!]$ by the matrix in the corresponding cell of $[\![M]\!]$.

BROADCAST ARITHMETIC

`P*=c` `(SO3partArray,Cscalar) -> SO3partArray`
`P*=c` `(SO3partArray,complex<float>) -> SO3partArray`
`P/=c` `(SO3partArray,Cscalar) -> SO3partArray`
`P/=c` `(SO3partArray,complex<float>) -> SO3partArray`

Multiply/divide $[\![P]\!]$ by the scalar c in-place.

`c*P` `(Cscalar,SO3partArray) -> SO3partArray`
`c*P` `(complex<float>,SO3partArray) -> SO3partArray`
`P*c` `(SO3partArray,Cscalar) -> SO3partArray`
`P*c` `(SO3partArray,complex<float>) -> SO3partArray`
`P/c` `(SO3partArray,Cscalar) -> SO3partArray`
`P/c` `(SO3partArray,complex<float>) -> SO3partArray`

Multiply/divide $[\![P]\!]$ by the scalar c.

`P1+P2` `(SO3part,SO3partArray) -> SO3partArray`
`P1+P2` `(SO3partArray,SO3part) -> SO3partArray`
`P1-P2` `(SO3part,SO3partArray) -> SO3partArray`
`P1-P2` `(SO3partArray,SO3part) -> SO3partArray`

Compute $P_1+[\![P_2]\!]$, $[\![P_1]\!]+P_2$, $P_1-[\![P_2]\!]$ or $[\![P_1]\!]-P_2$.

`M*P` `(Ctensor,SO3partArray) -> SO3partArray`

Multiply each cell of $[\![P]\!]$ by the matrix M.

SCATTERING ARITHMETIC

`scatter(C)*P` `(Ctensor,SO3partArray) -> SO3partArray`
`P*scatter(C)` `(SO3partArray,Ctensor) -> SO3partArray`
`P/scatter(C)` `(SO3partArray,Ctensor) -> SO3partArray`

Multiply/divide each cell of $[\![P]\!]$ by the scalar in the corresponding entry of C.

`P*=scatter(C)` `(SO3partArray,Ctensor) -> SO3partArray`
`P/=scatter(C)` `(SO3partArray,Ctensor) -> SO3partArray`

Multiply/divide each cell of $[\![P]\!]$ by the scalar in the corresponding entry of C in-place.

MEMBER FUNCTIONS

`SO3partArray rotate(const SO3element& R) const`
Rotate each cell by $R \in \mathrm{SO}(3)$.

OTHER FUNCTIONS

`norm2(P)` `SO3partArray -> Ctensor`
The tensor of squared Frobenius norms $\|P\|^2_{\mathrm{Frob}}$.

`inp(P1,P2)` `(SO3partArray,SO3partArray) -> Ctensor`
The tensor of inner products $\langle P_1, P_2 \rangle$.

CG-PRODUCTS

`CGproduct(P1,P2,l)` `(SO3partArray,SO3partArray,int) -> SO3partArray`
Compute the ℓ'th component of the Clebsch–Gordan product, $[P_1 \otimes_{cg} P_2]_\ell$, cell-by-cell.

`CGproduct(P1,P2,l)` `(SO3part,SO3partArray,int) -> SO3partArray`
`CGproduct(P1,P2,l)` `(SO3partArray,SO3part,int) -> SO3partArray`
Compute the ℓ'th component of the Clebsch–Gordan product by broadcasting P_1 resp. P_2.

`outerprod<CGproduct>(P1,P2,l)` `(SO3partArray,SO3partArray,int) -> SO3partArray`
`matrixprod<CGproduct>(P1,P2,l)` `(SO3partArray,SO3partArray,int) -> SO3partArray`
`convolution<CGproduct>(P1,P2,l)` `(SO3partArray,SO3partArray,int) -> SO3partArray`
Compute the outer product $[\![P_1]\!]$ and $[\![P_2]\!]$, matrix-like product, or convolution, where the elementary products are Clebsch–Gordan products.

I/O

`string str(const indent="") const`
Print `P` to string with the optional indentation `indent`.

SO3vecArray

An `SO3vecArray` $[\![V]\!]$ is an array of `SO3vec` objects.

Derived from: `cnine::ArrayPack<SO3partArray>`

CONSTRUCTORS

`SO3vecArray(const Gdims& adims, const SO3type& tau, [NBU], [FILLTYPE], [DEVICE])`
Create an arrray of `SO3vec` objects. The array dimensions are specified in `adims`. `FILLTYPE` can be `fill::raw` (default), `fill::zero` or `fill::gaussian`.

`SO3vecArray(const Gdims& adims, const SO3vec& v, [DEVICE])`
Create an `SO3vecArray` in which each cell is a copy of `v`.

STATIC CONSTRUCTORS

`SO3vecArray SO3vecArray::zero(const Gdims& adims, const SO3type& tau, [NBU], [DEVICE])`
`SO3vecArray SO3vecArray::ones(const Gdims& adims, const SO3type& tau, [NBU], [DEVICE])`
`SO3vecArray SO3vecArray::gaussian(const Gdims& adims, const SO3type& tau, [NBU], [DEVICE])`
Create a new `SO3vecArray` on device `DEVICE`.

VARIANT CONSTRUCTORS

`SO3vecArray(const SO3vecArray& P, const device& dev)`
Create a copy of $[\![V]\!]$ on device `dev`.

MEMBER ACCESS

`int getL() const`
Return L, the highest ℓ index.

`SO3type get_tau() const`
Return τ, the type of $\mathbf{v}$.

`int get_nbu() const`
Return the bundle dimension. If bundling is off, returns -1.

`int get_dev() const`
Return the device that this `SO3vecArray` is stored on.

`Gdims get_adims()`
Return the array dimensions.

`SO3vec get_cell(int i1,...,int ik) const`
Return the cell $[\![V]\!](i_1,\ldots,i_k)$.

`CtensorArray& set_cell(int i1,...,int ik, const SO3vec& A)`
Set $[\![V]\!](i_1,\ldots,i_k)$=A.

`SO3vec get_cell(Gindex& aix) const`
Return the cell $[\![V]\!](i_1,\ldots,i_k)$.

`CtensorArray& set_cell(Gindex& aix, const SO3vec& A)`
Set $[\![V]\!](i_1,\ldots,i_k) = A$.

`SO3vec cell(const Gindex& aix)` ☜
Return a temporary `SO3vec` capturing cell `aix`.

`SO3partArray get_part(int l)`
`SO3vec& set_part(int, l, const SO3partArray& P)`
Get/set the array of ℓ'th isotypic parts.

`SO3partArray get(int l)` ☜
Return a temporary to the array of ℓ'th isotypic parts.

ARRAY OPERATIONS

`SO3vecArray broaden(int ix, int n) const`
Create a $d+1$ dimensional array by stacking n copies of $[\![V]\!]$ along dimension ix.

`SO3vecArray reduce(int ix) const`
Create a $d-1$ dimensional array by summing out dimension `ix`.

`SO3vec reduce() const`
Reduce $[\![V]\!]$ by summing all entries into a single `SO3vec`.

`reshape(const Gdims& adims)`
Reinterpret $[\![V]\!]$ as an array of dimensions `adims`.

CELL-BY-CELL ARITHMETIC

`V1+V2` `(SO3vecArray,SO3vecArray) -> SO3vecArray`
`V1-V2` `(SO3vecArray,SO3vecArray) -> SO3vecArray`

Compute $[\![V_1]\!]+[\![V_2]\!]$ resp. $[\![V_1]\!]-[\![V_2]\!]$.

`W*V` `(CtensorPackArray,SO3vecArray) -> SO3vecArray`

Multiply each cell of $[\![V]\!]$ by the `CtensorPack` in the corresponding cell of $[\![W]\!]$.

BROADCASTING ARITHMETIC

`c*V` `(Cscalar,SO3vecArray) -> SO3vecArray`
`c*V` `(complex<float>,SO3vecArray) -> SO3vecArray`
`V*c` `(SO3vecArray,Cscalar) -> SO3vecArray`
`V*c` `(SO3vecArray,complex<float>) -> SO3vecArray`
`V/c` `(SO3vecArray,Cscalar) -> SO3vecArray`
`V/c` `(SO3vecArray,complex<float>) -> SO3vecArray`

Multiply/divide $[\![V]\!]$ by the scalar c.

`V1+V2` `(SO3vec,SO3vecArray) -> SO3vecArray`
`V1+V2` `(SO3vecArray,SO3vec) -> SO3vecArray`
`V1-V2` `(SO3vec,SO3vecArray) -> SO3vecArray`
`V1-V2` `(SO3vecArray,SO3vec) -> SO3vecArray`

Compute $P_1+[\![P_2]\!]$, $[\![P_1]\!]+P_2$, $P_1-[\![P_2]\!]$ or $[\![P_1]\!]-P_2$.

`W*V` `(CtensorPack,SO3vecArray) -> SO3vecArray`

Multiply each cell of $[\![V]\!]$ by the `CtensorArray` W.

SCATTERING ARITHMETIC

`scatter(C)*V` `(Ctensor,SO3vecArray) -> SO3vecArray`
`V*scatter(C)` `(SO3vecArray,Ctensor) -> SO3vecArray`
`V/scatter(C)`

Multiply/divide each cell of $[\![V]\!]$ by the scalar in the corresponding entry of C.

IN-PLACE OPERATIONS

`+=V2` `(SO3vecArray,SO3vecArray)->SO3vecArray`
`-=V2` `(SO3vecArray,SO3vecArray)->SO3vecArray`

Increment/decrement $[\![V]\!]$ by $[\![V_2]\!]$.

MEMBER FUNCTIONS

`SO3vecArray rotate(const SO3element& R) const`
Rotate each cell by $R \in \mathrm{SO}(3)$.

OTHER FUNCTIONS

`norm2(V)` `SO3vecArray -> Ctensor`
The tensor of squared Frobenius norms $\|V\|^2_{\mathrm{Frob}}$.

`inp(V1,V2)` `(SO3vecArray,SO3vecArray) -> Ctensor`
The tensor of inner products $\langle V_1, V_2 \rangle$.

CG-PRODUCTS

`CGproduct(V1,V2,[lmax])` `(SO3vecArray,SO3vecArray,int) -> SO3vecArray`
Compute the Clebsch–Gordan product, $V_1 \otimes_{cg} V_2$, up to band limit $\ell_{\max}$ cell-by-cell.

`CGproduct(V1,V2,[lmax])` `(SO3vec,SO3vecArray,int) -> SO3vecArray`
`CGproduct(V1,V2,[lmax])` `(SO3vecArray,SO3vec,int) -> SO3vecArray`
Compute the ℓ'th component of the Clebsch–Gordan product by broadcasting V_1 resp. V_2.

`Outer<CGproduct>(V1,V2,[lmax])` `(SO3vecArray,SO3vecArray,int) -> SO3vecArray`
`Mprod<CGproduct>(V1,V2,[lmax])` `(SO3vecArray,SO3vecArray,int) -> SO3vecArray`
`Convolve<CGproduct>(V1,V2,[lmax])` `(SO3vecArray,SO3vecArray,int) -> SO3vecArray`
Compute the outer product of $[\![V_1]\!]$ and $[\![V_2]\!]$, matrix-like product, or convolution, where the elementary products are Clebsch–Gordan products.

I/O

`string str(const indent="") const`
Print $[\![V]\!]$ to string with the optional indentation `indent`.