```julia
# Tensor factorization
#----------------------
const OFA = OrthogonalFactorizationAlgorithm

import LinearAlgebra: svd!, svd
const SVDAlg = Union{SVD, SDD}

Base.@deprecate(
    svd(t::AbstractTensorMap, leftind::IndexTuple, rightind::IndexTuple;
            trunc::TruncationScheme = notrunc(), p::Real = 2, alg::SVDAlg = SDD()),
    tsvd(t, leftind, rightind; trunc = trunc, p = p, alg = alg))
Base.@deprecate(
    svd(t::AbstractTensorMap;
            trunc::TruncationScheme = notrunc(), p::Real = 2, alg::SVDAlg = SDD()),
    tsvd(t; trunc = trunc, p = p, alg = alg))
Base.@deprecate(
    svd!(t::AbstractTensorMap;
            trunc::TruncationScheme = notrunc(), p::Real = 2, alg::SVDAlg = SDD()),
    tsvd(t; trunc = trunc, p = p, alg = alg))

"""
    tsvd(t::AbstractTensorMap, leftind::Tuple, rightind::Tuple;
        trunc::TruncationScheme = notrunc(), p::Real = 2, alg::Union{SVD, SDD} =
SDD())
        -> U, S, V, ϵ

Compute the (possibly truncated)) singular value decomposition such that
`norm(permute(t, leftind, rightind) - U * S *V) ≈ ϵ`, where `ϵ` thus represents
the truncation error.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in
`t` to be destroyed/overwritten, by using `tsvd!(t, trunc = notrunc(), p = 2)`.

A truncation parameter `trunc` can be specified for the new internal dimension, in
which
case a truncated singular value decomposition will be computed. Choices are:
*   `notrunc()`: no truncation (default);
*   `truncerr(η::Real)`: truncates such that the p-norm of the truncated singular
values is
    smaller than `η` times the p-norm of all singular values;
*   `truncdim(χ::Int)`: truncates such that the equivalent total dimension of the
internal
    vector space is no larger than `χ`;
*   `truncspace(V)`: truncates such that the dimension of the internal vector
space is
    smaller than that of `V` in any sector.
*   `trunbelow(χ::Real)`: truncates such that every singular value is larger then
`χ` ;

The method `tsvd` also returns the truncation error `ϵ`, computed as the `p` norm
of the
```

```
singular values that were truncated.

THe keyword `alg` can be equal to `SVD()` or `SDD()`, corresponding to the
underlying LAPACK
algorithm that computes the decomposition (`_gesvd` or `_gesdd`).

Orthogonality requires `spacetype(t)<:InnerProductSpace`, and `svd(!)` is currently
only implemented for `spacetype(t)<:EuclideanSpace`.
"""
tsvd(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple; kwargs...) =
    tsvd!(permute(t, p1, p2; copy = true); kwargs...)

"""
    leftorth(t::AbstractTensorMap, leftind::Tuple, rightind::Tuple;
                alg::OrthogonalFactorizationAlgorithm = QRpos()) -> Q, R

Create orthonormal basis `Q` for indices in `leftind`, and remainder `R` such that
`permute(t, leftind, rightind) = Q*R`.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in `t`
to be destroyed/overwritten, by using `leftorth!(t, alg = QRpos())`.

Different algorithms are available, namely `QR()`, `QRpos()`, `SVD()` and
`Polar()`. `QR()`
and `QRpos()` use a standard QR decomposition, producing an upper triangular
matrix `R`.
`Polar()` produces a Hermitian and positive semidefinite `R`. `QRpos()` corrects
the
standard QR decomposition such that the diagonal elements of `R` are positive. Only
`QRpos()` and `Polar()` are uniqe (no residual freedom) so that they always return
the same
result for the same input tensor `t`.

Orthogonality requires `spacetype(t)<:InnerProductSpace`, and `leftorth(!)` is
currently
only implemented for `spacetype(t)<:EuclideanSpace`.
"""
leftorth(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple; kwargs...) =
    leftorth!(permute(t, p1, p2; copy = true); kwargs...)

"""
    rightorth(t::AbstractTensorMap, leftind::Tuple, rightind::Tuple;
                alg::OrthogonalFactorizationAlgorithm = LQpos()) -> L, Q

Create orthonormal basis `Q` for indices in `rightind`, and remainder `L` such that
`permute(t, leftind, rightind) = L*Q`.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in `t`
```

```julia
to be destroyed/overwritten, by using `rightorth!(t, alg = LQpos())`.

Different algorithms are available, namely `LQ()`, `LQpos()`, `RQ()`, `RQpos()`,
`SVD()` and
`Polar()`. `LQ()` and `LQpos()` produce a lower triangular matrix `L` and are
computed using
a QR decomposition of the transpose. `RQ()` and `RQpos()` produce an upper
triangular
remainder `L` and only works if the total left dimension is smaller than or equal
to the
total right dimension. `LQpos()` and `RQpos()` add an additional correction such
that the
diagonal elements of `L` are positive. `Polar()` produces a Hermitian and positive
semidefinite `L`. Only `LQpos()`, `RQpos()` and `Polar()` are uniqe (no residual
freedom) so
that they always return the same result for the same input tensor `t`.

Orthogonality requires `spacetype(t)<:InnerProductSpace`, and `rightorth(!)` is
currently
only implemented for `spacetype(t)<:EuclideanSpace`.
"""
rightorth(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple; kwargs...) =
    rightorth!(permute(t, p1, p2; copy = true); kwargs...)

"""
    leftnull(t::AbstractTensor, leftind::Tuple, rightind::Tuple;
                alg::OrthogonalFactorizationAlgorithm = QRpos()) -> N

Create orthonormal basis for the orthogonal complement of the support of the
indices in
`leftind`, such that `N' * permute(t, leftind, rightind) = 0`.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in `t`
to be destroyed/overwritten, by using `leftnull!(t, alg = QRpos())`.

Different algorithms are available, namely `QR()` (or equivalently, `QRpos()`),
`SVD()` and
`SDD()`. The first assumes that the matrix is full rank and requires
`iszero(atol)` and
`iszero(rtol)`. With `SVD()` and `SDD()`, `rightnull` will use the corresponding
singular
value decomposition, and one can specify an absolute or relative tolerance for
which
singular values are to be considered zero, where `max(atol, norm(t)*rtol)` is used
as upper
bound.

Orthogonality requires `spacetype(t)<:InnerProductSpace`, and `leftnull(!)` is
currently
only implemented for `spacetype(t)<:EuclideanSpace`.
"""
```

```
leftnull(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple; kwargs...) =
    leftnull!(permute(t, p1, p2; copy = true); kwargs...)

"""
    rightnull(t::AbstractTensor, leftind::Tuple, rightind::Tuple;
                alg::OrthogonalFactorizationAlgorithm = LQ(),
                atol::Real = 0.0,
                rtol::Real = eps(real(float(one(eltype(t)))))*iszero(atol)) -> N

Create orthonormal basis for the orthogonal complement of the support of the
indices in
`rightind`, such that `permute(t, leftind, rightind)*N' = 0`.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in `t`
to be destroyed/overwritten, by using `rightnull!(t, alg = LQpos())`.

Different algorithms are available, namely `LQ()` (or equivalently, `LQpos`),
`SVD()` and
`SDD()`. The first assumes that the matrix is full rank and requires
`iszero(atol)` and
`iszero(rtol)`. With `SVD()` and `SDD()`, `rightnull` will use the corresponding
singular
value decomposition, and one can specify an absolute or relative tolerance for
which
singular values are to be considered zero, where `max(atol, norm(t)*rtol)` is used
as upper
bound.

Orthogonality requires `spacetype(t)<:InnerProductSpace`, and `rightnull(!)` is
currently
only implemented for `spacetype(t)<:EuclideanSpace`.
"""
rightnull(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple; kwargs...) =
    rightnull!(permute(t, p1, p2; copy = true); kwargs...)

"""
    eigen(t::AbstractTensor, leftind::Tuple, rightind::Tuple; kwargs...) -> D, V

Compute eigenvalue factorization of tensor `t` as linear map from `rightind` to
`leftind`.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in `t`
to be destroyed/overwritten, by using `eigen!(t)`. Note that the permuted tensor
on which
`eigen!` is called should have equal domain and codomain, as otherwise the
eigenvalue
decomposition is meaningless and cannot satisfy
```

````julia
permute(t, leftind, rightind) * V = V * D
```

Accepts the same keyword arguments `scale`, `permute` and `sortby` as `eigen` of
dense
matrices. See the corresponding documentation for more information.

See also `eig` and `eigh`
"""
LinearAlgebra.eigen(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple;
kwargs...) =
    eigen!(permute(t, p1, p2; copy = true); kwargs...)

"""
    eig(t::AbstractTensor, leftind::Tuple, rightind::Tuple; kwargs...) -> D, V

Compute eigenvalue factorization of tensor `t` as linear map from `rightind` to
`leftind`.
The function `eig` assumes that the linear map is not hermitian and returns type
stable
complex valued `D` and `V` tensors for both real and complex valued `t`. See
`eigh` for
hermitian linear maps

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in
`t` to be destroyed/overwritten, by using `eig!(t)`. Note that the permuted tensor
on
which `eig!` is called should have equal domain and codomain, as otherwise the
eigenvalue
decomposition is meaningless and cannot satisfy
```
permute(t, leftind, rightind) * V = V * D
```

Accepts the same keyword arguments `scale`, `permute` and `sortby` as `eigen` of
dense matrices. See the corresponding documentation for more information.

See also `eigen` and `eigh`.
"""
eig(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple; kwargs...) =
    eig!(permute(t, p1, p2; copy = true); kwargs...)

"""
    eigh(t::AbstractTensorMap{<:EuclideanSpace}, leftind::Tuple, rightind::Tuple)
-> D, V

Compute eigenvalue factorization of tensor `t` as linear map from `rightind` to
`leftind`.
The function `eigh` assumes that the linear map is hermitian and `D` and `V`
tensors with
the same `eltype` as `t`. See `eig` and `eigen` for non-hermitian tensors.
````
````

```
Hermiticity
requires that the tensor acts on inner product spaces, and the current
implementation
requires `spacetyp(t) <: EuclideanSpace`.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in
`t` to be destroyed/overwritten, by using `eigh!(t)`. Note that the permuted
tensor on
which `eigh!` is called should have equal domain and codomain, as otherwise the
eigenvalue
decomposition is meaningless and cannot satisfy
```
permute(t, leftind, rightind) * V = V * D
```

See also `eigen` and `eig`.
"""
eigh(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple) =
    eigh!(permute(t, p1, p2; copy = true))

"""
    isposdef(t::AbstractTensor{<:EuclideanSpace}, leftind::Tuple, rightind::Tuple)
-> ::Bool

Test whether a tensor `t` is positive definite as linear map from `rightind` to
`leftind`.

If `leftind` and `rightind` are not specified, the current partition of left and
right
indices of `t` is used. In that case, less memory is allocated if one allows the
data in
`t` to be destroyed/overwritten, by using `isposdef!(t)`. Note that the permuted
tensor on
which `isposdef!` is called should have equal domain and codomain, as otherwise it
is
meaningless

Accepts the same keyword arguments `scale`, `permute` and `sortby` as `eigen` of
dense
matrices. See the corresponding documentation for more information.
"""
LinearAlgebra.isposdef(t::AbstractTensorMap, p1::IndexTuple, p2::IndexTuple) =
    isposdef!(permute(t, p1, p2; copy = true))

tsvd(t::AbstractTensorMap; trunc::TruncationScheme = NoTruncation(),
                            p::Real = 2, alg::Union{SVD, SDD} = SDD()) =
    tsvd!(copy(t); trunc = trunc, p = p, alg = alg)
leftorth(t::AbstractTensorMap; alg::OFA = QRpos(), kwargs...) =
    leftorth!(copy(t); alg = alg, kwargs...)
rightorth(t::AbstractTensorMap; alg::OFA = LQpos(), kwargs...) =
    rightorth!(copy(t); alg = alg, kwargs...)
```

```
leftnull(t::AbstractTensorMap; alg::OFA = QR(), kwargs...) =
    leftnull!(copy(t); alg = alg, kwargs...)
rightnull(t::AbstractTensorMap; alg::OFA = LQ(), kwargs...) =
    rightnull!(copy(t); alg = alg, kwargs...)
LinearAlgebra.eigen(t::AbstractTensorMap; kwargs...) = eigen!(copy(t); kwargs...)
eig(t::AbstractTensorMap; kwargs...) = eig!(copy(t); kwargs...)
eigh(t::AbstractTensorMap; kwargs...) = eigh!(copy(t); kwargs...)
LinearAlgebra.isposdef(t::AbstractTensorMap) = isposdef!(copy(t))

# Orthogonal factorizations (mutation for recycling memory):
# only correct if Euclidean inner product
#-----------------------------------------------------------------------------------
--------
leftorth!(t::AdjointTensorMap{S}; alg::OFA = QRpos()) where {S<:EuclideanSpace} =
    map(adjoint, reverse(rightorth!(adjoint(t); alg = alg')))

rightorth!(t::AdjointTensorMap{S}; alg::OFA = LQpos()) where {S<:EuclideanSpace} =
    map(adjoint, reverse(leftorth!(adjoint(t); alg = alg')))

leftnull!(t::AdjointTensorMap{S}; alg::OFA = QR(), kwargs...) where
{S<:EuclideanSpace} =
    adjoint(rightnull!(adjoint(t); alg = alg', kwargs...))

rightnull!(t::AdjointTensorMap{S}; alg::OFA = LQ(), kwargs...) where
{S<:EuclideanSpace} =
    adjoint(leftnull!(adjoint(t); alg = alg', kwargs...))

function tsvd!(t::AdjointTensorMap{S};
                trunc::TruncationScheme = NoTruncation(),
                p::Real = 2,
                alg::Union{SVD, SDD} = SDD()) where {S<:EuclideanSpace}
    u, s, vt, err = tsvd!(adjoint(t); trunc = trunc, p = p, alg = alg)
    return adjoint(vt), adjoint(s), adjoint(u), err
end

function leftorth!(t::TensorMap{<:EuclideanSpace};
                    alg::Union{QR, QRpos, QL, QLpos, SVD, SDD, Polar} = QRpos(),
                    atol::Real = zero(float(real(eltype(t)))),
                    rtol::Real = (alg ∉ (SVD(), SDD())) ?
zero(float(real(eltype(t)))) :
                    eps(real(float(one(eltype(t)))))*iszero(atol))
    if !iszero(rtol)
        atol = max(atol, rtol*norm(t))
    end
    I = sectortype(t)
    S = spacetype(t)
    A = storagetype(t)
    Qdata = SectorDict{I, A}()
    Rdata = SectorDict{I, A}()
    dims = SectorDict{I, Int}()
    for c in blocksectors(domain(t))
        isempty(block(t,c)) && continue
        Q, R = _leftorth!(block(t, c), alg, atol)
        Qdata[c] = Q
```

```
        Rdata[c] = R
        dims[c] = size(Q, 2)
    end
    V = S(dims)
    if alg isa Polar
        @assert V ≅ domain(t)
        W = domain(t)
    elseif length(domain(t)) == 1 && domain(t) ≅ V
        W = domain(t)
    elseif length(codomain(t)) == 1 && codomain(t) ≅ V
        W = codomain(t)
    else
        W = ProductSpace(V)
    end
    return TensorMap(Qdata, codomain(t)←W), TensorMap(Rdata, W←domain(t))
end

function leftnull!(t::TensorMap{<:EuclideanSpace};
                    alg::Union{QR, QRpos, SVD, SDD} = QRpos(),
                    atol::Real = zero(float(real(eltype(t)))),
                    rtol::Real = (alg ∉ (SVD(), SDD())) ?
zero(float(real(eltype(t)))) :
                    eps(real(float(one(eltype(t)))))*iszero(atol))
    if !iszero(rtol)
        atol = max(atol, rtol*norm(t))
    end
    I = sectortype(t)
    S = spacetype(t)
    A = storagetype(t)
    V = codomain(t)
    Ndata = SectorDict{I, A}()
    dims = SectorDict{I, Int}()
    for c in blocksectors(V)
        N = _leftnull!(block(t, c), alg, atol)
        Ndata[c] = N
        dims[c] = size(N, 2)
    end
    W = S(dims)
    return TensorMap(Ndata, V←W)
end

function rightorth!(t::TensorMap{<:EuclideanSpace};
                    alg::Union{LQ, LQpos, RQ, RQpos, SVD, SDD, Polar} = LQpos(),
                    atol::Real = zero(float(real(eltype(t)))),
                    rtol::Real = (alg ∉ (SVD(), SDD())) ?
zero(float(real(eltype(t)))) :
                    eps(real(float(one(eltype(t)))))*iszero(atol))
    if !iszero(rtol)
        atol = max(atol, rtol*norm(t))
    end
    I = sectortype(t)
    S = spacetype(t)
    A = storagetype(t)
    Ldata = SectorDict{I, A}()
```

```
    Qdata = SectorDict{I, A}()
    dims = SectorDict{I, Int}()
    for c in blocksectors(codomain(t))
        isempty(block(t,c)) && continue
        L, Q = _rightorth!(block(t, c), alg, atol)
        Ldata[c] = L
        Qdata[c] = Q
        dims[c] = size(Q, 1)
    end
    V = S(dims)
    if alg isa Polar
        @assert V ≅ codomain(t)
        W = codomain(t)
    elseif length(codomain(t)) == 1 && codomain(t) ≅ V
        W = codomain(t)
    elseif length(domain(t)) == 1 && domain(t) ≅ V
        W = domain(t)
    else
        W = ProductSpace(V)
    end
    return TensorMap(Ldata, codomain(t)←W), TensorMap(Qdata, W←domain(t))
end

function rightnull!(t::TensorMap{<:EuclideanSpace};
                        alg::Union{LQ, LQpos, SVD, SDD} = LQpos(),
                        atol::Real = zero(float(real(eltype(t)))),
                        rtol::Real = (alg ∉ (SVD(), SDD())) ?
zero(float(real(eltype(t)))) :
                        eps(real(float(one(eltype(t)))))*iszero(atol))
    if !iszero(rtol)
        atol = max(atol, rtol*norm(t))
    end
    I = sectortype(t)
    S = spacetype(t)
    A = storagetype(t)
    V = domain(t)
    Ndata = SectorDict{I, A}()
    dims = SectorDict{I, Int}()
    for c in blocksectors(V)
        N = _rightnull!(block(t, c), alg, atol)
        Ndata[c] = N
        dims[c] = size(N, 1)
    end
    W = S(dims)
    return TensorMap(Ndata, W←V)
end

function tsvd!(t::TensorMap{<:EuclideanSpace};
                    trunc::TruncationScheme = NoTruncation(),
                    p::Real = 2,
                    alg::Union{SVD, SDD} = SDD())
    S = spacetype(t)
    I = sectortype(t)
    A = storagetype(t)
```

```
    Ar = similarstoragetype(t, real(eltype(t)))
    Udata = SectorDict{I, A}()
    Σmdata = SectorDict{I, Ar}() # this will contain the singular values as matrix
    Vdata = SectorDict{I, A}()
    dims = SectorDict{sectortype(t), Int}()
    if isempty(blocksectors(t))
        W = S(dims)
        truncerr = zero(real(eltype(t)))
        return TensorMap(Udata, codomain(t)←W), TensorMap(Σmdata, W←W),
                    TensorMap(Vdata, W←domain(t)), truncerr
    end
    for (c, b) in blocks(t)
        U, Σ, V = _svd!(b, alg)
        Udata[c] = U
        Vdata[c] = V
        if @isdefined Σdata # cannot easily infer the type of Σ, so use this
            construction
            Σdata[c] = Σ
        else
            Σdata = SectorDict(c=>Σ)
        end
        dims[c] = length(Σ)
    end
    if !isa(trunc, NoTruncation)
        Σdata, truncerr = _truncate!(Σdata, trunc, p)
        truncdims = SectorDict{I, Int}()
        for c in blocksectors(t)
            truncdim = length(Σdata[c])
            if truncdim != 0
                truncdims[c] = truncdim
                if truncdim != dims[c]
                    Udata[c] = Udata[c][:, 1:truncdim]
                    Vdata[c] = Vdata[c][1:truncdim, :]
                end
            else
                delete!(Udata, c)
                delete!(Vdata, c)
                delete!(Σdata, c)
            end
        end
        dims = truncdims
        W = S(dims)
    else
        W = S(dims)
        if length(domain(t)) == 1 && domain(t)[1] ≅ W
            W = domain(t)[1]
        elseif length(codomain(t)) == 1 && codomain(t)[1] ≅ W
            W = codomain(t)[1]
        end
        truncerr = abs(zero(eltype(t)))
    end
    for (c, Σ) in Σdata
        Σmdata[c] = copyto!(similar(Σ, length(Σ), length(Σ)), Diagonal(Σ))
    end
```

```julia
    return TensorMap(Udata, codomain(t)←W), TensorMap(Σmdata, W←W),
            TensorMap(Vdata, W←domain(t)), truncerr
end

function LinearAlgebra.ishermitian(t::TensorMap)
    domain(t) == codomain(t) || return false
    spacetype(t) <: EuclideanSpace || return false # hermiticity only defined for
        euclidean
    for (c, b) in blocks(t)
        ishermitian(b) || return false
    end
    return true
end

LinearAlgebra.eigen!(t::TensorMap) = ishermitian(t) ? eigh!(t) : eig!(t)

function eigh!(t::TensorMap{<:EuclideanSpace}; kwargs...)
    domain(t) == codomain(t) ||
        throw(SpaceMismatch("`eigh!` requires domain and codomain to be the same"))
    S = spacetype(t)
    I = sectortype(t)
    A = storagetype(t)
    Ar = similarstoragetype(t, real(eltype(t)))
    Ddata = SectorDict{I, Ar}()
    Vdata = SectorDict{I, A}()
    dims = SectorDict{I, Int}()
    for (c, b) in blocks(t)
        values, vectors = eigen!(Hermitian(b); kwargs...)
        d = length(values)
        Ddata[c] = copyto!(similar(values, (d, d)), Diagonal(values))
        Vdata[c] = vectors
        dims[c] = d
    end
    if length(domain(t)) == 1
        W = domain(t)[1]
    else
        W = S(dims)
    end
    return TensorMap(Ddata, W←W), TensorMap(Vdata, domain(t)←W)
end

function eig!(t::TensorMap; kwargs...)
    domain(t) == codomain(t) ||
        throw(SpaceMismatch("`eig!` requires domain and codomain to be the same"))
    S = spacetype(t)
    I = sectortype(t)
    T = complex(eltype(t))
    Ac = similarstoragetype(t, T)
    Ddata = SectorDict{I, Ac}()
    Vdata = SectorDict{I, Ac}()
    dims = SectorDict{I, Int}()
    for (c, b) in blocks(t)
        values, vectors = eigen!(b; kwargs...)
        d = length(values)
```

```
        Ddata[c] = copyto!(similar(values, T, (d, d)), Diagonal(values))
        if eltype(vectors) == T
            Vdata[c] = vectors
        else
            Vdata[c] = copyto!(similar(vectors, T), vectors)
        end
        dims[c] = d
    end
    if length(domain(t)) == 1
        W = domain(t)[1]
    else
        W = S(dims)
    end
    return TensorMap(Ddata, W←W), TensorMap(Vdata, domain(t)←W)
end

function LinearAlgebra.isposdef!(t::TensorMap)
    domain(t) == codomain(t) ||
        throw(SpaceMismatch("`isposdef` requires domain and codomain to be the
same"))
    spacetype(t) <: EuclideanSpace || return false
    for (c, b) in blocks(t)
        isposdef!(b) || return false
    end
    return true
end
```