Physlib Monthly Report

1–31 March 2026

Contributors: Alex Meiburg, Bolton Bailey, Daniel Morrison, devtest-boop, Gabe, Gregory Loges, ijsvogelberg, JayYarlott, Jeremias Rieser, Joseph Tooby-Smith, Leonardo A. Lessa, Leonardo Lessa, Pietro Monticone, Rodolfo R. Soldati, Ruizsolveall, Tanner Duve, Zhi Kai Pong

Reviewers: Alex Meiburg, Daniel Morrison, Joseph Tooby-Smith, Zhi Kai Pong

Generated 14 August 2026 from commit `f2fd447`.

Abstract

Physlib is a community library of formalised physics for the Lean 4 proof assistant: definitions and results from across physics, stated in a formal language and checked by machine. This report is an automatically generated record of one month's additions to it.

It covers 1–31 March 2026 on branch `master` of `leanprover-community/physlib`, between commits `6286c6b` and `f2fd447`. Over that period 17 contributors landed 546 commits, changing 40,754 lines across 500 files and adding 2,247 new Lean declarations, each catalogued with its statement in Section 2. The complete diff is available at GitHub compare.

Contents

1 Introduction **2**
- 1.1 Contributors . . . 2
- 1.2 Reviewers . . . 2
- 1.3 Modified subfolders . . . 2
- 1.4 New declarations by kind . . . 3

2 Details by file **4**
- 2.1 `PhysLean.ClassicalMechanics.DampedHarmonicOscillator.Basic` . . . 4
- 2.2 `PhysLean.Electromagnetism.ThreeDimension.Basic` . . . 4
- 2.3 `PhysLean.Mathematics.KroneckerDelta` . . . 5
- 2.4 `PhysLean.QuantumMechanics.DDimensions.Operators.Commutation` . . . 6
- 2.5 `PhysLean.QuantumMechanics.DDimensions.Operators.Momentum` . . . 6
- 2.6 `PhysLean.QuantumMechanics.DDimensions.Operators.Position` . . . 7
- 2.7 `PhysLean.QuantumMechanics.DDimensions.Operators.Unbounded` . . . 8
- 2.8 `PhysLean.QuantumMechanics.DDimensions.SpaceDHilbertSpace.SchwartzSubmodule` 12
- 2.9 `PhysLean.Relativity.Tensors.Elab` . . . 13
- 2.10 `PhysLean.SpaceAndTime.Space.Basic` . . . 14
- 2.11 `PhysLean.SpaceAndTime.Space.Derivatives.Basic` . . . 15
- 2.12 `PhysLean.SpaceAndTime.Space.Derivatives.Curl` . . . 16
- 2.13 `PhysLean.SpaceAndTime.Space.Derivatives.Grad` . . . 16
- 2.14 `PhysLean.SpaceAndTime.Space.Integrals.Basic` . . . 17
- 2.15 `PhysLean.SpaceAndTime.Space.Integrals.NormPow` . . . 18
- 2.16 `PhysLean.SpaceAndTime.Space.Integrals.RadialAngularMeasure` . . . 19

2.17 PhysLean.SpaceAndTime.Space.Module . . . 19
2.18 PhysLean.SpaceAndTime.Time.Derivatives . . . 21
2.19 QuantumInfo.ClassicalInfo.Capacity . . . 21
2.20 QuantumInfo.ClassicalInfo.Channel . . . 22
2.21 QuantumInfo.ClassicalInfo.Distribution . . . 23
2.22 QuantumInfo.ClassicalInfo.Entropy . . . 28
2.23 QuantumInfo.ClassicalInfo.ForMathlib.Analysis.SpecialFunctions.Log.NegMulLog 29
2.24 QuantumInfo.ClassicalInfo.Prob . . . 30
2.25 QuantumInfo.Finite.AxiomatizedEntropy.Defs . . . 38
2.26 QuantumInfo.Finite.AxiomatizedEntropy.Renyi . . . 40
2.27 QuantumInfo.Finite.Braket . . . 41
2.28 QuantumInfo.Finite.Capacity . . . 46
2.29 QuantumInfo.Finite.Channel.DegradableOrder . . . 48
2.30 QuantumInfo.Finite.CPTPMap.Bundled . . . 48
2.31 QuantumInfo.Finite.CPTPMap.CPTP . . . 55
2.32 QuantumInfo.Finite.CPTPMap.Dual . . . 64
2.33 QuantumInfo.Finite.CPTPMap.MatrixMap . . . 68
2.34 QuantumInfo.Finite.CPTPMap.Unbundled . . . 72
2.35 QuantumInfo.Finite.Distance.Fidelity . . . 80
2.36 QuantumInfo.Finite.Distance.TraceDistance . . . 81
2.37 QuantumInfo.Finite.Ensemble . . . 81
2.38 QuantumInfo.Finite.Entanglement . . . 86
2.39 QuantumInfo.Finite.Entropy.DPI . . . 89
2.40 QuantumInfo.Finite.Entropy.Relative . . . 93
2.41 QuantumInfo.Finite.Entropy.SSA . . . 108
2.42 QuantumInfo.Finite.Entropy.VonNeumann . . . 119
2.43 QuantumInfo.Finite.MState . . . 123
2.44 QuantumInfo.Finite.Pinching . . . 138
2.45 QuantumInfo.Finite.POVM . . . 141
2.46 QuantumInfo.Finite.Qubit.Basic . . . 142
2.47 QuantumInfo.Finite.ResourceTheory.FreeState . . . 146
2.48 QuantumInfo.Finite.ResourceTheory.HypothesisTesting . . . 152
2.49 QuantumInfo.Finite.ResourceTheory.ResourceTheory . . . 154
2.50 QuantumInfo.Finite.ResourceTheory.SteinsLemma . . . 155
2.51 QuantumInfo.Finite.Unitary . . . 162
2.52 QuantumInfo.ForMathlib.ContinuousLinearMap . . . 163
2.53 QuantumInfo.ForMathlib.ContinuousSup . . . 164
2.54 QuantumInfo.ForMathlib.Filter . . . 167
2.55 QuantumInfo.ForMathlib.HermitianMat.Basic . . . 167
2.56 QuantumInfo.ForMathlib.HermitianMat.CFC . . . 180
2.57 QuantumInfo.ForMathlib.HermitianMat.Inner . . . 191
2.58 QuantumInfo.ForMathlib.HermitianMat.Jordan . . . 196
2.59 QuantumInfo.ForMathlib.HermitianMat.LogExp . . . 198
2.60 QuantumInfo.ForMathlib.HermitianMat.NonSingular . . . 203
2.61 QuantumInfo.ForMathlib.HermitianMat.Order . . . 207
2.62 QuantumInfo.ForMathlib.HermitianMat.Proj . . . 213
2.63 QuantumInfo.ForMathlib.HermitianMat.Reindex . . . 219
2.64 QuantumInfo.ForMathlib.HermitianMat.Rpow . . . 220
2.65 QuantumInfo.ForMathlib.HermitianMat.Schatten . . . 226
2.66 QuantumInfo.ForMathlib.HermitianMat.Sqrt . . . 228
2.67 QuantumInfo.ForMathlib.HermitianMat.Trace . . . 229

2.68 QuantumInfo.ForMathlib.HermitianMat.Unitary . . . 233
2.69 QuantumInfo.ForMathlib.IsMaximalSelfAdjoint . . . 234
2.70 QuantumInfo.ForMathlib.Isometry . . . 234
2.71 QuantumInfo.ForMathlib.Lieb . . . 240
2.72 QuantumInfo.ForMathlib.LimSupInf . . . 240
2.73 QuantumInfo.ForMathlib.LinearEquiv . . . 242
2.74 QuantumInfo.ForMathlib.Majorization . . . 244
2.75 QuantumInfo.ForMathlib.Matrix . . . 250
2.76 QuantumInfo.ForMathlib.MatrixNorm.TraceNorm . . . 260
2.77 QuantumInfo.ForMathlib.Minimax . . . 262
2.78 QuantumInfo.ForMathlib.Misc . . . 264
2.79 QuantumInfo.ForMathlib.SionMinimax . . . 265
2.80 QuantumInfo.ForMathlib.Superadditive . . . 270
2.81 QuantumInfo.ForMathlib.ULift . . . 270
2.82 QuantumInfo.ForMathlib.Unitary . . . 271
2.83 QuantumInfo.Regularized . . . 272
2.84 QuantumInfo.StatMech.Hamiltonian . . . 274
2.85 QuantumInfo.StatMech.IdealGas . . . 275
2.86 QuantumInfo.StatMech.ThermoQuantities . . . 275

1 Introduction

During Mar 2026, 17 contributors submitted 546 commits that changed 40,754 lines (+37,415 / -3,339) across 500 files spanning 33 top-level areas of the repository. Of those, 89 files were newly added and 2 were removed outright. This report catalogues 2,247 newly added Lean declarations: 1,297 theorems, 417 lemmas, 318 defs, 167 instances, 20 structures, 15 abbrevs, 11 classs, 1 opaque, and 1 axiom. We're delighted to recognize the following first-time contributors this month: Alex Meiburg, Bolton Bailey, devtest-boop, Gabe, ijsvogelberg, JayYarlott, Jeremias Rieser, Leonardo A. Lessa, Leonardo Lessa, Rodolfo R. Soldati, Ruizsolveall, and Tanner Duve.

See the full compare diff and the list of commits on GitHub for the raw source.

1.1 Contributors

The following individuals authored commits merged during this reporting period, listed alphabetically.

Contributor	Lines Changed	Commits
Alex Meiburg	91,184	372
Bolton Bailey	157	3
Daniel Morrison	82	1
devtest-boop	82	5
Gabe	578	2
Gregory Loges	2,593	11
ijsvogelberg	4	1
JayYarlott	48	1
Jeremias Rieser	192	6
Joseph Tooby-Smith	3,355	29
Leonardo A. Lessa	2,143	31
Leonardo Lessa	1,561	43
Pietro Monticone	84	3
Rodolfo R. Soldati	746	19
Ruizsolveall	80	1
Tanner Duve	17	2
Zhi Kai Pong	4,835	16

1.2 Reviewers

The following individuals approved pull requests during this reporting period, listed alphabetically.

Reviewer	PRs Reviewed
Alex Meiburg	1
Daniel Morrison	11
Joseph Tooby-Smith	17
Zhi Kai Pong	6

1.3 Modified subfolders

Subfolder	Files	New	Deleted	Additions	Deletions

QuantumInfo/Finite	31	+31	–	+14,555	-0
QuantumInfo/ForMathlib	34	+34	–	+12,440	-0
PhysLean/SpaceAndTime	32	+4	-1	+1,933	-1,047
PhysLean/QuantumMechanics	29	–	–	+1,397	-779
QuantumInfo	4	+4	–	+2,015	-0
QuantumInfo/ClassicalInfo	6	+6	–	+1,119	-0
PhysLean.lean	1	–	–	+410	-404
PhysLean/Relativity	81	–	–	+520	-154
PhysLean/QFT	72	–	–	+446	-153
PhysLean/Mathematics	33	–	–	+409	-116
PhysLean/Particles	64	–	–	+372	-113
QuantumInfo/StatMech	3	+3	–	+442	-0
PhysLean/ClassicalMechanics	18	–	–	+275	-146
PhysLean/Electromagnetism	23	+1	–	+206	-62
PhysLean/Meta	15	–	–	+128	-61
README.md	1	–	–	+105	-67
lakefile.toml	1	–	-1	+0	-157
lakefile.lean	1	+1	–	+117	-0
PhysLean/StringTheory	13	–	–	+75	-25
PhysLean/Units	14	–	–	+77	-21
docs	2	+2	–	+86	-0
.github/workflows	1	+1	–	+82	-0
QuantumInfo/InfiniteDim	1	+1	–	+74	-0
PhysLean/StatisticalMechanics	5	–	–	+33	-13
QuantumInfo.lean	1	+1	–	+42	-0
PhysLean/Thermodynamics	4	–	–	+23	-7
PhysLean/CondensedMatter	2	–	–	+11	-5
PhysLean/Cosmology	2	–	–	+9	-3
PhysLean/Optics	2	–	–	+8	-2
.codespellignore	1	–	–	+3	-0
.gitignore	1	–	–	+1	-2
lake-manifest.json	1	–	–	+1	-1
scripts	1	–	–	+1	-1

1.4 New declarations by kind

Kind	Count
theorem	1,297
lemma	417
def	318
instance	167
structure	20
abbrev	15
class	11
opaque	1
axiom	1

2 Details by file

2.1 PhysLean.ClassicalMechanics.DampedHarmonicOscillator.Basic

PhysLean/ClassicalMechanics/DampedHarmonicOscillator/Basic.lean on GitHub.

lemma kineticEnergy_eq [source]

Equalties for energy eqns

```
lemma kineticEnergy_eq (x: Time → ℝ) :
    kineticEnergy S x = fun t => (1 / 2 : ℝ) * S.m * (Time.deriv x t)^2
```

lemma potentialEnergy_eq [source]

```
lemma potentialEnergy_eq (x: Time → ℝ) :
    potentialEnergy S x = fun t => (1 / 2 : ℝ) * S.k * (x t)^2
```

lemma energy_dissipation_rate [source]

Derives the energy dissipation rate from the equation of motion

```
lemma energy_dissipation_rate (x: Time → ℝ) (h1 : S.EquationOfMotion x)
    (hx : ContDiff ℝ ∞ x) :
    Time.deriv (energy S x) t = - S.γ * (Time.deriv x t)^2
```

lemma energy_not_conserved [source]

```
lemma energy_not_conserved (x: Time → ℝ) (h1 : S.EquationOfMotion x)
    (hx : ContDiff ℝ ∞ x)
    (hdx : Time.deriv x t ≠ 0)
    (hγ : 0 < S.γ) :
    Time.deriv (energy S x) t < 0
```

2.2 PhysLean.Electromagnetism.ThreeDimension.Basic

PhysLean/Electromagnetism/ThreeDimension/Basic.lean on GitHub.

theorem electricField_eq_3D [source]

The electric field written in terms of the scalar and vector potentials as `- ∇ φ - ∂ₜ A`.

```
theorem electricField_eq_3D :
    E = fun t x => - ∇ (φ t) x - ∂ₜ (fun t => A t x) t
```

theorem magneticField_eq_3D [source]

The magnetic field written as the curl of the vector potential as `∇ × A`.

```
theorem magneticField_eq_3D :
    B = fun t x => (∇ × (A t)) x
```

2.3 PhysLean.Mathematics.KroneckerDelta

PhysLean/Mathematics/KroneckerDelta.lean on GitHub.

lemma eq_one_of_same [source]

```
@[simp]
lemma eq_one_of_same (i : α) : δ[i,i] = 1
```

lemma eq_zero_of_ne [source]

```
lemma eq_zero_of_ne {i j : α} (h : i ≠ j) : δ[i,j] = 0
```

lemma eq_of_coe [source]

```
@[simp]
lemma eq_of_coe {p : α → Prop} (i j : Subtype p) : δ[(i : α),j] = δ[i,j]
```

lemma eq_zero_of_not [source]

```
lemma eq_zero_of_not {p : α → Prop} {i j : α} (hi : ¬p i) (hj : p j) : δ[i,j] = 0
```

lemma smul_of_eq_zero [source]

```
lemma smul_of_eq_zero [AddMonoid M] (i j : α) {f : α → α → M} (hf : f i i = 0) :
    δ[i,j] • f i j = 0
```

lemma smul_eq_zero_iff [source]

```
lemma smul_eq_zero_iff [AddMonoid M] (i j : α) (f : α → α → M) :
    δ[i,j] • f i j = 0 ↔ i ≠ j ∨ f i i = 0
```

lemma smul_eq_zero_iff' [source]

```
lemma smul_eq_zero_iff' [AddMonoid M] (i : α) (f : α → α → M) :
    (∀ j : α, δ[i,j] • f i j = 0) ↔ f i i = 0
```

lemma smul_eq_zero_iff'' [source]

```
lemma smul_eq_zero_iff'' [AddMonoid M] (f : α → α → M) :
    (∀ i j : α, δ[i,j] • f i j = 0) ↔ ∀ i : α, f i i = 0
```

lemma symm [source]

```
lemma symm (i j : α) : δ[i,j] = δ[j,i]
```

lemma smul_symm [source]

```
lemma smul_symm [AddMonoid M] (i j : α) (f : α → α → M) : δ[i,j] • f j i = δ[i,j] • f i j
```

lemma symmetrize [source]

```
lemma symmetrize [AddMonoid M] (i j : α) (f : α → α → M) :
    δ[i,j] • (f i j + f j i) = (2 * δ[i,j]) • f i j
```

lemma symmetrize' [source]

```
lemma symmetrize' [AddCommMonoid M] {K : Type*} [Semifield K] [CharZero K] [Module K M]
    (i j : α) (f : α → α → M) : δ[i,j] • (2 : K)⁻¹ • (f i j + f j i) = δ[i,j] • f i j
```

lemma smul_sub_eq_zero [source]

```
@[simp]
lemma smul_sub_eq_zero [AddGroup M] (i j : α) (f : α → α → M) : δ[i,j] • (f i j - f j i) = 0
```

lemma sum_mul [source]

```
@[simp]
lemma sum_mul [Fintype α] (i j : α) : ∑ k : α, δ[i,k] * δ[k,j] = δ[i,j]
```

lemma sum_smul [source]

```
@[simp]
lemma sum_smul [Fintype α] (i : α) (f : α → M) : ∑ j : α, δ[i,j] • f j = f i
```

lemma sum_sum_smul_eq_zero [source]

```
lemma sum_sum_smul_eq_zero [Fintype α] {f : α → α → M} (hf : ∀ i : α, f i i = 0) :
    ∑ i : α, ∑ j : α, δ[i,j] • f i j = 0
```

lemma finset_sum_smul [source]

```
lemma finset_sum_smul (s : Finset α) (i : α) (f : α → M) :
    ∑ j ∈ s, δ[i,j] • f j = if i ∈ s then f i else 0
```

lemma finset_sum_sum_smul_eq_zero [source]

```
lemma finset_sum_sum_smul_eq_zero {s s' : Finset α} {f : α → α → M}
    (hf : ∀ i ∈ s ∩ s', f i i = 0) : ∑ i ∈ s, ∑ j ∈ s', δ[i,j] • f i j = 0
```

2.4 PhysLean.QuantumMechanics.DDimensions.Operators.Commutation

PhysLean/QuantumMechanics/DDimensions/Operators/Commutation.lean on GitHub.

lemma angularMomentum_comp_antisymm_sum [source]

```
private lemma angularMomentum_comp_antisymm_sum {d : ℕ} (a b : Fin d) :
    ∑ x : Fin d, 𝐋[x,a] ∘L 𝐋[b,x] = ∑ x : Fin d, 𝐋[a,x] ∘L 𝐋[x,b]
```

2.5 PhysLean.QuantumMechanics.DDimensions.Operators.Momentum

PhysLean/QuantumMechanics/DDimensions/Operators/Momentum.lean on GitHub.

def momentumOperatorSchwartz [source]

The momentum operators defined on the Schwartz submodule.

```
def momentumOperatorSchwartz : schwartzSubmodule d →ₗ[ℂ] schwartzSubmodule d
```

lemma momentumOperatorSchwartz_isSymmetric [source]

```
lemma momentumOperatorSchwartz_isSymmetric :
    (momentumOperatorSchwartz i).IsSymmetric
```

def momentumUnboundedOperator [source]

The symmetric momentum unbounded operators with domain the Schwartz submodule of the Hilbert space.

```
def momentumUnboundedOperator :
    UnboundedOperator (SpaceDHilbertSpace d) (SpaceDHilbertSpace d)
```

2.6 PhysLean.QuantumMechanics.DDimensions.Operators.Position

PhysLean/QuantumMechanics/DDimensions/Operators/Position.lean on GitHub.

lemma norm_sq_add_unit_sq_pos [source]

```
lemma norm_sq_add_unit_sq_pos {d : ℕ} (ε : ℝˣ) (x : Space d) : 0 < ‖x‖ ^ 2 + ε ^ 2
```

lemma normRegularizedPow_pos [source]

```
lemma normRegularizedPow_pos (d : ℕ) (ε : ℝˣ) (s : ℝ) (x : Space d) :
    0 < normRegularizedPow d ε s x
```

def positionOperatorSchwartz [source]

The position operators defined on the Schwartz submodule.

```
def positionOperatorSchwartz : schwartzSubmodule d →ₗ[ℂ] schwartzSubmodule d
```

lemma positionOperatorSchwartz_isSymmetric [source]

```
lemma positionOperatorSchwartz_isSymmetric : (positionOperatorSchwartz i).IsSymmetric
```

def positionUnboundedOperator [source]

The symmetric position unbounded operators with domain the Schwartz submodule of the Hilbert space.

```
def positionUnboundedOperator : UnboundedOperator (SpaceDHilbertSpace d) (SpaceDHilbertSpace d)
```

def radiusRegPowOperatorSchwartz [source]

The (regularized) radius operators defined on the Schwartz submodule.

```
def radiusRegPowOperatorSchwartz {d : ℕ} (ε : ℝˣ) (s : ℝ) :
    schwartzSubmodule d →ₗ[ℂ] schwartzSubmodule d
```

lemma radiusRegPowOperatorSchwartz_isSymmetric [source]

```
lemma radiusRegPowOperatorSchwartz_isSymmetric {d : ℕ} (ε : ℝˣ) (s : ℝ) :
    (radiusRegPowOperatorSchwartz (d := d) ε s).IsSymmetric
```

def radiusRegPowUnboundedOperator [source]

The symmetric (regularized) radius unbounded operators with domain the Schwartz submodule of the Hilbert space.

```
def radiusRegPowUnboundedOperator {d : ℕ} (ε : ℝˣ) (s : ℝ) :
    UnboundedOperator (SpaceDHilbertSpace d) (SpaceDHilbertSpace d)
```

2.7 PhysLean.QuantumMechanics.DDimensions.Operators.Unbounded

PhysLean/QuantumMechanics/DDimensions/Operators/Unbounded.lean on GitHub.

lemma ext [source]

```
@[ext]
lemma ext {U₁ U₂ : UnboundedOperator H H'} (h : U₁.toLinearPMap = U₂.toLinearPMap) :
    U₁ = U₂
```

instance CoeFun (UnboundedOperator H H') (fun U ↦ U.domain → H') [source]

```
instance : CoeFun (UnboundedOperator H H') (fun U ↦ U.domain → H')
```

lemma inner_map_polarization [source]

```
lemma inner_map_polarization {T : UnboundedOperator H H} (x y : T.domain) :
    ⟪T x, ↑y⟫_ℂ = (⟪T (x + y), ↑(x + y)⟫_ℂ - ⟪T (x - y), ↑(x - y)⟫_ℂ
    - I * ⟪T (x + I • y), ↑(x + I • y)⟫_ℂ + I * ⟪T (x - I • y), ↑(x - I • y)⟫_ℂ) / 4
```

lemma inner_map_polarization' [source]

```
lemma inner_map_polarization' {T : UnboundedOperator H H} (x y : T.domain) :
    ⟪↑x, T y⟫_ℂ = (⟪↑(x + y), T (x + y)⟫_ℂ - ⟪↑(x - y), T (x - y)⟫_ℂ
    - I * ⟪↑(x + I • y), T (x + I • y)⟫_ℂ + I * ⟪↑(x - I • y), T (x - I • y)⟫_ℂ) / 4
```

instance PartialOrder (UnboundedOperator H H') [source]

```
instance : PartialOrder (UnboundedOperator H H') where
```

lemma isClosable_of_zero [source]

```
lemma isClosable_of_zero {E F : Type*} [NormedAddCommGroup E] [InnerProductSpace ℂ E]
    [NormedAddCommGroup F] [InnerProductSpace ℂ F] {f : LinearPMap ℂ E F} (hf : f.toFun' = 0) :
    f.IsClosable
```

instance Zero (UnboundedOperator H H') [source]

```
instance : Zero (UnboundedOperator H H')
```

lemma zero_toLinearPMap [source]

```
@[simp]
lemma zero_toLinearPMap : (0 : UnboundedOperator H H').toLinearPMap = 0
```

instance Inhabited (UnboundedOperator H H') [source]

```
instance : Inhabited (UnboundedOperator H H')
```

instance Add (UnboundedOperator H H') [source]

```
noncomputable instance : Add (UnboundedOperator H H') where
```

lemma add_domain_of_dense [source]

The domain of $U₁ + U₂$ is $D(U₁) ∩ D(U₂)$ when it is dense.

```
lemma add_domain_of_dense {U₁ U₂ : UnboundedOperator H H'}
    (hD : Dense (U₁.domain ⊓ U₂.domain : Set H)) : (U₁ + U₂).domain = U₁.domain ⊓ U₂.domain
```

lemma add_domain_of_not_dense [source]

The junk value for $U₁ + U₂$ has domain all of H when $D(U₁) ∩ D(U₂)$ is not dense.

```
lemma add_domain_of_not_dense {U₁ U₂ : UnboundedOperator H H'}
    (hD : ¬Dense (U₁.domain ⊓ U₂.domain : Set H)) : (U₁ + U₂).domain = ⊤
```

lemma mem_domain_of_dense [source]

```
lemma mem_domain_of_dense {U₁ U₂ : UnboundedOperator H H'}
    (hD : Dense (U₁.domain ⊓ U₂.domain : Set H)) (ψ : (U₁ + U₂).domain) :
    ↑ψ ∈ U₁.domain ∧ ↑ψ ∈ U₂.domain
```

lemma add_toLinearPMap_of_dense_closable [source]

$U₁ + U₂$ is the unbounded operator corresponding to $U₁.toLinearPMap + U₂.toLinearPMap$, provided it is both densely defined and closable.

```
lemma add_toLinearPMap_of_dense_closable {U₁ U₂ : UnboundedOperator H H'}
    (hD : Dense (U₁.domain ⊓ U₂.domain : Set H))
    (hC : (U₁.toLinearPMap + U₂.toLinearPMap).IsClosable) :
    (U₁ + U₂).toLinearPMap = U₁.toLinearPMap + U₂.toLinearPMap
```

lemma add_toLinearPMap_of_dense_not_closable [source]

The junk value for $U_1 + U_2$ when $D(U_1) \cap D(U_2)$ is dense and $U_1 + U_2$ is not closable.

```
lemma add_toLinearPMap_of_dense_not_closable {U₁ U₂ : UnboundedOperator H H'}
    (hD : Dense (U₁.domain ⊓ U₂.domain : Set H))
    (hC : ¬(U₁.toLinearPMap + U₂.toLinearPMap).IsClosable) :
    (U₁ + U₂).toLinearPMap = ⟨U₁.domain ⊓ U₂.domain, 0⟩
```

lemma add_toLinearPMap_of_not_dense [source]

The junk value for $U_1 + U_2$ when $D(U_1) \cap D(U_2)$ is not dense.

```
lemma add_toLinearPMap_of_not_dense {U₁ U₂ : UnboundedOperator H H'}
    (hD : ¬Dense (U₁.domain ⊓ U₂.domain : Set H)) : (U₁ + U₂).toLinearPMap = 0
```

lemma add_apply_of_dense_closable [source]

$(U_1 + U_2)\psi = U_1\psi + U_2\psi$ provided $U_1 + U_2$ is not given by a junk value.

```
lemma add_apply_of_dense_closable {U₁ U₂ : UnboundedOperator H H'}
    (hD : Dense (U₁.domain ⊓ U₂.domain : Set H))
    (hC : (U₁.toLinearPMap + U₂.toLinearPMap).IsClosable) (ψ : (U₁ + U₂).domain) :
    (U₁ + U₂) ψ = U₁ ⟨ψ, (mem_domain_of_dense hD ψ).1⟩ + U₂ ⟨ψ, (mem_domain_of_dense hD ψ).2⟩
```

instance AddZeroClass (UnboundedOperator H H') [source]

```
noncomputable instance : AddZeroClass (UnboundedOperator H H') where
```

lemma add_assoc [source]

Addition of unbounded operators is associative when neither of the intermediate additions, $U_1 + U_2$ and $U_2 + U_3$, is given by a junk value.

```
lemma add_assoc {U₁ U₂ U₃ : UnboundedOperator H H'}
    (hD₁₂ : Dense (U₁.domain ⊓ U₂.domain : Set H)) (hC₁₂ : (U₁.1 + U₂.1).IsClosable)
    (hD₂₃ : Dense (U₂.domain ⊓ U₃.domain : Set H)) (hC₂₃ : (U₂.1 + U₃.1).IsClosable) :
    U₁ + U₂ + U₃ = U₁ + (U₂ + U₃)
```

lemma smul_mem_graph_of_mem_smul_graph [source]

```
lemma smul_mem_graph_of_mem_smul_graph {E F : Type*} [NormedAddCommGroup E] [InnerProductSpace
    ℂ E]
    [NormedAddCommGroup F] [InnerProductSpace ℂ F] {f : LinearPMap ℂ E F} {c : ℂ} (hc : c ≠ 0)
    {x : E × F} (h : x ∈ (c • f).graph.topologicalClosure) :
    (x.fst, c⁻¹ • x.snd) ∈ f.graph.topologicalClosure
```

lemma smul_isClosable_of_isClosable [source]

```
lemma smul_isClosable_of_isClosable {E F : Type*} [NormedAddCommGroup E] [InnerProductSpace ℂ E
    ]
    [NormedAddCommGroup F] [InnerProductSpace ℂ F] {f : LinearPMap ℂ E F} (hf : f.IsClosable)
    (c : ℂ) : (c • f).IsClosable
```

instance SMul ℂ (UnboundedOperator H H') [source]

```
noncomputable instance : SMul ℂ (UnboundedOperator H H') where
```

lemma smul_domain [source]

```
@[simp]
lemma smul_domain (c : ℂ) : (c • U).domain = U.domain
```

lemma smul_toLinearPMap [source]

```
@[simp]
lemma smul_toLinearPMap (c : ℂ) : (c • U).toLinearPMap = c • U.toLinearPMap
```

lemma zero_smul_le_zero [source]

```
lemma zero_smul_le_zero : (0 : ℂ) • U ≤ 0
```

instance DistribSMul ℂ (UnboundedOperator H H') [source]

```
noncomputable instance : DistribSMul ℂ (UnboundedOperator H H') where
```

lemma le_closure [source]

```
lemma le_closure : U ≤ U.closure
```

lemma isClosed_def [source]

```
lemma isClosed_def : IsClosed U ↔ U.closure = U
```

lemma adjoint_dense_of_isClosable [source]

The adjoint of a densely defined, closable `LinearPMap` is densely defined.

```
lemma adjoint_dense_of_isClosable {f : LinearPMap ℂ H H'} (h_dense : Dense (f.domain : Set H))
    (h_closable : f.IsClosable) : Dense (f†.domain : Set H')
```

lemma adjoint_closure_eq_adjoint [source]

```
lemma adjoint_closure_eq_adjoint : U†.closure = U†
```

lemma le_adjoint_adjoint [source]

```
lemma le_adjoint_adjoint : U ≤ U††
```

lemma isClosed_iff [source]

```
lemma isClosed_iff : IsClosed U ↔ U†† = U
```

lemma adjoint_ge_adjoint_of_le [source]

```
lemma adjoint_ge_adjoint_of_le {U₁ U₂ : UnboundedOperator H H'} (h : U₁ ≤ U₂) : U₂† ≤ U₁†
```

lemma closure_mono [source]

```
lemma closure_mono {U₁ U₂ : UnboundedOperator H H'} (h : U₁ ≤ U₂) : U₁.closure ≤ U₂.closure
```

def IsSymmetric [source]

T is symmetric if ⟪T x, y⟫ = ⟪x, T y⟫ for all x,y ∈ T.domain.

```
def IsSymmetric : Prop
```

lemma isSymmetric_iff_inner_map_self_real [source]

```
lemma isSymmetric_iff_inner_map_self_real :
    IsSymmetric T ↔ ∀ x : T.domain, (starRingEnd ℂ) ⟪T x, x⟫_ℂ = ⟪T x, x⟫_ℂ
```

lemma isSymmetric_iff_le_adjoint [source]

```
lemma isSymmetric_iff_le_adjoint : IsSymmetric T ↔ T ≤ T†
```

lemma isClosed_of_isSelfAdjoint [source]

```
lemma isClosed_of_isSelfAdjoint {T : UnboundedOperator H H} (hT : IsSelfAdjoint T) : IsClosed T
```

lemma isSymmetric_of_isSelfAdjoint [source]

```
lemma isSymmetric_of_isSelfAdjoint {T : UnboundedOperator H H} (hT : IsSelfAdjoint T) :
    IsSymmetric T
```

def IsEssentiallySelfAdjoint [source]

T is essentially self-adjoint if its closure is self-adjoint.

```
def IsEssentiallySelfAdjoint : Prop
```

lemma isEssentiallySelfAdjoint_def [source]

```
lemma isEssentiallySelfAdjoint_def : IsEssentiallySelfAdjoint T ↔ T† = T.closure
```

lemma isSelfAdjoint_isEssentiallySelfAdjoint [source]

```
lemma isSelfAdjoint_isEssentiallySelfAdjoint {T : UnboundedOperator H H} (hT : IsSelfAdjoint T)
     :
    IsEssentiallySelfAdjoint T
```

2.8 PhysLean.QuantumMechanics.DDimensions.SpaceDHilbertSpace.SchwartzSubmodule

PhysLean/QuantumMechanics/DDimensions/SpaceDHilbertSpace/SchwartzSubmodule.lean on GitHub.

instance CoeFun (schwartzSubmodule d) fun _ ↦ Space d → ℂ [source]

```
instance : CoeFun (schwartzSubmodule d) fun _ ↦ Space d → ℂ
```

lemma val_eq_coe [source]

```
@[simp]
lemma val_eq_coe (ψ : schwartzSubmodule d) (x : Space d) : ψ.val x = ψ x
```

def schwartzEquiv [source]

The linear equivalence between the Schwartz functions `𝓢(Space d, ℂ)` and the Schwartz submodule of `SpaceDHilbertSpace d`.

```
def schwartzEquiv : 𝓢(Space d, ℂ) ≃ₗ[ℂ] schwartzSubmodule d
```

lemma schwartzEquiv_coe_ae [source]

```
lemma schwartzEquiv_coe_ae : (schwartzEquiv f) =ᵐ[volume] f
```

lemma schwartzEquiv_inner [source]

```
lemma schwartzEquiv_inner :
    ⟪schwartzEquiv f, schwartzEquiv g⟫_ℂ = ∫ x : Space d, starRingEnd ℂ (f x) * g x
```

lemma schwartzEquiv_ae_eq [source]

```
lemma schwartzEquiv_ae_eq (h : schwartzEquiv f =ᵐ[volume] schwartzEquiv g) : f = g
```

2.9 PhysLean.Relativity.Tensors.Elab

PhysLean/Relativity/Tensors/Elab.lean on GitHub.

def indexExprIsBracketEval [source]

Bool which is true if an index is evaluated bracket `[μ]`.

```
def indexExprIsBracketEval(stx : Syntax) : Bool
```

def getEvalBracketPos [source]

For list of `indexExpr` e.g. `[α, 3, β, 2, [γ]]`, `getEvalPos` returns a list of pairs `ℕ × Term` related to indices which are evaluated e.g. `[μ]`. The second element of each pair is the value corresponding to that index. The first element is the position of that number in the list of indices when all other numbered indices before it are removed. Thus for the example given `getEvalBracketPos` outputs `[(4, γ)]`.

```
def getEvalBracketPos (ind : List (TSyntax `indexExpr)) : TermElabM (List (ℕ × Term))
```

def evalTermBracketMap [source]

Given a list `l` of pairs `ℕ × Term` and a term `T` corresponding to a tensor tree, for each `(a, b)` in `l`, `evalSyntax` applies `TensorTree.eval a b` to `T` recursively. Here `a` is the position of the index to be evaluated and `b` is the value it is evaluated to from the `[μ]` syntax.

For example, if `l` is `[(1, μ), (1, ν)]` and `T` is a tensor tree then `evalSyntax l T` is `TensorTree.eval 1 ν (TensorTree.eval 1 μ T)`.

The list `l` is expected to be the output of `getEvalBracketPos`.

```
def evalTermBracketMap (l : List (ℕ × Term)) (T : Term) : Term
```

2.10 PhysLean.SpaceAndTime.Space.Basic

PhysLean/SpaceAndTime/Space/Basic.lean on GitHub.

instance Nonempty (Space d) [source]

```
instance {d} : Nonempty (Space d)
```

instance VSub (EuclideanSpace ℝ (Fin d)) (Space d) [source]

```
noncomputable instance {d} : VSub (EuclideanSpace ℝ (Fin d)) (Space d) where
```

lemma vsub_apply [source]

```
@[simp]
lemma vsub_apply {d} (s1 s2 : Space d) (i : Fin d) :
    (s1 -ᵥ s2) i = s1 i - s2 i
```

instance AddTorsor (EuclideanSpace ℝ (Fin d)) (Space d) [source]

```
noncomputable instance {d} : AddTorsor (EuclideanSpace ℝ (Fin d)) (Space d) where
```

instance Dist (Space d) [source]

```
noncomputable instance {d} : Dist (Space d) where
```

instance PseudoMetricSpace (Space d) [source]

```
noncomputable instance {d} : PseudoMetricSpace (Space d) where
```

instance NormedAddTorsor (EuclideanSpace ℝ (Fin d)) (Space d) [source]

```
noncomputable instance {d} : NormedAddTorsor (EuclideanSpace ℝ (Fin d)) (Space d) where
```

instance MetricSpace (Space d) [source]

```
noncomputable instance {d} : MetricSpace (Space d) where
```

instance Nontrivial (Space d.succ) [source]

```
instance {d : ℕ} : Nontrivial (Space d.succ) where
```

def manifoldStructure [source]

The manifold structure on `Space d`.

```
noncomputable def manifoldStructure (d : ℕ) :
    ModelWithCorners ℝ (EuclideanSpace ℝ (Fin d)) (Space d) where
```

lemma manifoldStructure_comp_manifoldStructure_symm [source]

```
@[simp]
lemma manifoldStructure_comp_manifoldStructure_symm {d : ℕ} :
    (↑(manifoldStructure d) ∘ ↑(manifoldStructure d).symm) = id
```

lemma manifoldStructure_comp_manifoldStructure_symm_apply [source]

```
lemma manifoldStructure_comp_manifoldStructure_symm_apply {d : ℕ}
    (x : EuclideanSpace ℝ (Fin d)) :
    (manifoldStructure d) ((manifoldStructure d).symm x) = x
```

lemma range_manifoldStructure [source]

```
@[simp]
lemma range_manifoldStructure {d : ℕ} :
    (Set.range ↑(manifoldStructure d)) = Set.univ
```

lemma contMDiff_vaddConst [source]

```
lemma contMDiff_vaddConst (d : ℕ) : ContMDiff
    (manifoldStructure d) (𝓘(ℝ, EuclideanSpace ℝ (Fin d)))  ⊤ (manifoldStructure d).toFun
```

2.11 PhysLean.SpaceAndTime.Space.Derivatives.Basic

PhysLean/SpaceAndTime/Space/Derivatives/Basic.lean on GitHub.

lemma deriv_eq_fderiv_fun [source]

```
lemma deriv_eq_fderiv_fun  [AddCommGroup M] [Module ℝ M] [TopologicalSpace M]
    (μ : Fin d) (f : Space d → M) :
    deriv μ f = fun x => fderiv ℝ (fun x => f x) x (basis μ)
```

lemma deriv_eq_mfderiv [source]

The spatial-derivative in terms of the derivative of functions between manifolds.

```
lemma deriv_eq_mfderiv {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (μ : Fin d) (f : Space d → M) (x : Space d) :
    deriv μ f x = mfderiv 𝓘(ℝ, Space d) 𝓘(ℝ, M) f x (basis μ)
```

lemma mdifferentiable_manifoldStructure_iff_differentiable [source]

```
lemma mdifferentiable_manifoldStructure_iff_differentiable {M d} [NormedAddCommGroup M]
    [NormedSpace ℝ M] {f : Space d → M} {x : Space d} :
    MDifferentiableAt (Space.manifoldStructure d) 𝓘(ℝ, M) f x ↔ DifferentiableAt ℝ f x
```

lemma deriv_eq_mfderiv_manifoldStructure [source]

The spatial-derivative in terms of the derivative of functions between manifolds with the manifold structure `Space.manifoldStructure d`.

```
lemma deriv_eq_mfderiv_manifoldStructure {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (μ : Fin d) (f : Space d → M) (x : Space d) :
    deriv μ f x = mfderiv (Space.manifoldStructure d) 𝓘(ℝ, M) f x (EuclideanSpace.single μ 1)
```

lemma deriv_contDiff [source]

```
lemma deriv_contDiff {d} {f : Space d → ℝ} (hf : ContDiff ℝ (n + 1) f) :
    ContDiff ℝ n fun x i => deriv i f x
```

2.12 PhysLean.SpaceAndTime.Space.Derivatives.Curl

PhysLean/SpaceAndTime/Space/Derivatives/Curl.lean on GitHub.

lemma curl_neg [source]

```
lemma curl_neg (f : Space → EuclideanSpace ℝ (Fin 3)) (hf : Differentiable ℝ f) :
    ∇ × (-f) = -∇ × f
```

lemma curl_sub [source]

```
lemma curl_sub (f1 f2 : Space → EuclideanSpace ℝ (Fin 3))
    (hf1 : Differentiable ℝ f1) (hf2 : Differentiable ℝ f2) :
    ∇ × (f1 - f2) = ∇ × f1 - ∇ × f2
```

lemma curl_of_grad_eq_zero [source]

```
lemma curl_of_grad_eq_zero (f : Space → ℝ) (hf : ContDiff ℝ 2 f) :
    ∇ × (∇ f) = 0
```

2.13 PhysLean.SpaceAndTime.Space.Derivatives.Grad

PhysLean/SpaceAndTime/Space/Derivatives/Grad.lean on GitHub.

lemma contDiff_grad [source]

```
@[fun_prop]
lemma contDiff_grad {n} {f : Space → ℝ} (hf : ContDiff ℝ (n + 1) f) :
    ContDiff ℝ n (fun x => ∇ f x)
```

def gradSchwartz [source]

The gradient on Scwhwartz maps.

```
noncomputable def gradSchwartz {d} : 𝓢(Space d, ℝ) →L[ℝ] 𝓢(Space d, EuclideanSpace ℝ (Fin d))
```

lemma gradSchwartz_apply_eq_grad [source]

```
lemma gradSchwartz_apply_eq_grad {d} (η : 𝓢(Space d, ℝ)) (x : Space d):
    gradSchwartz η x = grad η x
```

2.14 PhysLean.SpaceAndTime.Space.Integrals.Basic

PhysLean/SpaceAndTime/Space/Integrals/Basic.lean on GitHub.

lemma volume_metricBall_three_real [source]

```
@[simp]
lemma volume_metricBall_three_real :
    (volume.real (Metric.ball (0 : Space 3) 1)) = 4 / 3 * Real.pi
```

lemma integral_one_dim_eq_integral_real [source]

```
lemma integral_one_dim_eq_integral_real {f : Space 1 → ℝ} :
    ∫ x, f x ∂volume = ∫ x, f (oneEquiv.symm x) ∂volume
```

lemma integral_volume_eq_spherical [source]

```
lemma integral_volume_eq_spherical (d : ℕ) (f : Space d.succ → F)
    [NormedAddCommGroup F] [NormedSpace ℝ F] :
    ∫ x, f x ∂volume = ∫ x, f (x.2.1 • x.1.1) ∂(volume (α := Space d.succ).toSphere.prod
        (Measure.volumeIoiPow (Module.finrank ℝ (Space d.succ) - 1)))
```

instance SFinite (@Measure.comap ↑(Set.Ioi 0) ℝ Subtype.instMeasurableSpace Real… [source]

```
instance : SFinite (@Measure.comap ↑(Set.Ioi 0) ℝ Subtype.instMeasurableSpace
        Real.measureSpace.toMeasurableSpace Subtype.val volume)
```

lemma lintegral_volume_eq_spherical [source]

```
lemma lintegral_volume_eq_spherical (d : ℕ) (f : Space d.succ → ENNReal) (hf : Measurable f) :
    ∫⁻ x, f x ∂volume = ∫⁻ x, f (x.2.1 • x.1.1) ∂(volume (α := Space d.succ).toSphere.prod
        (Measure.volumeIoiPow (Module.finrank ℝ (Space d.succ) - 1)))
```

lemma lintegral_volume_eq_spherical_mul [source]

```
lemma lintegral_volume_eq_spherical_mul (d : ℕ) (f : Space d.succ → ENNReal) (hf : Measurable f
    ) :
    ∫⁻ x, f x ∂volume = ∫⁻ x, f (x.2.1 • x.1.1) * .ofReal (x.2.1 ^ d)
      ∂(volume (α := Space d.succ).toSphere.prod (Measure.volumeIoiPow 0))
```

2.15 PhysLean.SpaceAndTime.Space.Integrals.NormPow

PhysLean/SpaceAndTime/Space/Integrals/NormPow.lean on GitHub.

lemma npow_indicator_rpow_eq [source]

```
private lemma npow_indicator_rpow_eq {n : ℕ} {s : Set ℝ} (hs : 0 ∉ s) (p : ℝ) :
    (fun r ↦ r ^ n • s.indicator (fun r ↦ r ^ p) r) = s.indicator (fun r ↦ r ^ (n + p))
```

lemma integrableOn_indicator_iff [source]

```
private lemma _root_.MeasureTheory.integrableOn_indicator_iff {s : Set ℝ} (hs : MeasurableSet s
    )
    (t : Set ℝ) (f : ℝ → ℝ) :
    IntegrableOn (s.indicator f) t volume ↔ IntegrableOn f (s ∩ t) volume
```

lemma integrableOn_norm_rpow_ball_iff [source]

The function $x \mapsto \|x\|^p$ is integrable on {x : Space d | 0 ≤ ‖x‖ < b} iff 0 < d + p.

```
lemma integrableOn_norm_rpow_ball_iff {d : ℕ} (hd : 0 < d) {b : ℝ} (hb : 0 < b) (p : ℝ) :
    IntegrableOn (fun x : Space d ↦ ‖x‖ ^ p) (Metric.ball 0 b) ↔ 0 < d + p
```

lemma integrableOn_norm_rpow_ball_compl_iff [source]

The function $x \mapsto \|x\|^p$ is integrable on {x : Space d | 0 < a ≤ ‖x‖} iff d + p < 0.

```
lemma integrableOn_norm_rpow_ball_compl_iff {d : ℕ} (hd : 0 < d) {a : ℝ} (ha : 0 < a) (p : ℝ) :
    IntegrableOn (fun x : Space d ↦ ‖x‖ ^ p) (Metric.ball 0 a)ᶜ ↔ d + p < 0
```

lemma integrableOn_norm_rpow_shell [source]

The function $x \mapsto \|x\|^p$ is integrable on the shell {x : Space d | 0 < a ≤ ‖x‖ ∧ ‖x‖ < b}.

```
lemma integrableOn_norm_rpow_shell {d : ℕ} {a : ℝ} (ha : 0 < a) (b p : ℝ) :
    IntegrableOn (fun x : Space d ↦ ‖x‖ ^ p) ((Metric.ball 0 b) ∩ (Metric.ball 0 a)ᶜ)
```

lemma integrableOn_norm_rpow_iff_of_isBounded_nhds [source]

The function $x \mapsto \|x\|^p$ is integrable on a bounded neighborhood of the origin iff 0 < d + p.

```
lemma integrableOn_norm_rpow_iff_of_isBounded_nhds {d : ℕ} (hd : 0 < d) {s : Set (Space d)}
    (hs : Bornology.IsBounded s) (hs' : s ∈ nhds 0) (p : ℝ) :
    IntegrableOn (fun x : Space d ↦ ‖x‖ ^ p) s ↔ 0 < d + p
```

lemma integrableOn_norm_rpow_of_isBounded_compl_nhds [source]

The function $x \mapsto \|x\|^p$ is integrable on a bounded subset with the origin in its exterior.

```
lemma integrableOn_norm_rpow_of_isBounded_compl_nhds {d : ℕ} {s : Set (Space d)}
    (hs : Bornology.IsBounded s) (hs' : sᶜ ∈ nhds 0) (p : ℝ) :
    IntegrableOn (fun x : Space d ↦ ‖x‖ ^ p) s
```

lemma integrableOn_norm_rpow_of_compl_nhds [source]

The function $x \mapsto \|x\|^p$ is integrable on a subset with the origin in its exterior provided the decay at infinity is fast enough.

```
lemma integrableOn_norm_rpow_of_compl_nhds {d : ℕ} (hd : 0 < d) {s : Set (Space d)}
    (hs : sᶜ ∈ nhds 0) {p : ℝ} (h : d + p < 0) : IntegrableOn (fun x : Space d ↦ ‖x‖ ^ p) s
```

2.16 PhysLean.SpaceAndTime.Space.Integrals.RadialAngularMeasure

PhysLean/SpaceAndTime/Space/Integrals/RadialAngularMeasure.lean on GitHub.

instance SFinite (radialAngularMeasure (d [source]

```
instance (d : ℕ) : SFinite (radialAngularMeasure (d := d))
```

lemma lintegral_radialMeasure [source]

```
lemma lintegral_radialMeasure {d : ℕ} (f : Space d → ENNReal) (hf : Measurable f) :
    ∫⁻ x, f x ∂radialAngularMeasure = ∫⁻ x, ENNReal.ofReal (1 / ‖x‖ ^ (d - 1)) * f x
```

lemma lintegral_radialMeasure_eq_spherical_mul [source]

```
lemma lintegral_radialMeasure_eq_spherical_mul (d : ℕ) (f : Space d.succ → ENNReal)
    (hf : Measurable f) :
    ∫⁻ x, f x ∂radialAngularMeasure = ∫⁻ x, f (x.2.1 • x.1.1)
      ∂(volume (α := Space d.succ).toSphere.prod (Measure.volumeIoiPow 0))
```

lemma radialAngularMeasure_closedBall [source]

```
@[simp]
lemma radialAngularMeasure_closedBall (r : ℝ) :
    radialAngularMeasure (Metric.closedBall (0 : Space 3) r) = ENNReal.ofReal (4 * π * r)
```

lemma radialAngularMeasure_real_closedBall [source]

```
lemma radialAngularMeasure_real_closedBall (r : ℝ) (hr : 0 < r) :
    radialAngularMeasure.real (Metric.closedBall (0 : Space 3) r) = 4 * π * r
```

lemma integrable_radialAngularMeasure_of_spherical [source]

```
lemma integrable_radialAngularMeasure_of_spherical {d : ℕ} (f : Space d.succ → F)
    (hae : StronglyMeasurable f)
    (hf : Integrable (fun x => f (x.2.1 • x.1.1))
    (volume (α := Space d.succ).toSphere.prod (Measure.volumeIoiPow 0))) :
    Integrable f radialAngularMeasure
```

2.17 PhysLean.SpaceAndTime.Space.Module

PhysLean/SpaceAndTime/Space/Module.lean on GitHub.

lemma dist_eq_norm [source]

```
@[simp]
lemma dist_eq_norm {d} (p q : Space d) :
    dist p q = ‖p - q‖
```

lemma contDiffOn_vadd [source]

```
@[fun_prop]
lemma contDiffOn_vadd (s : Space d) :
    ContDiffOn ℝ ω (fun (v : EuclideanSpace ℝ (Fin d)) => v +ᵥ s) Set.univ
```

lemma vadd_differentiable [source]

```
@[fun_prop]
lemma vadd_differentiable {d} (s : Space d) :
    Differentiable ℝ (fun (v : EuclideanSpace ℝ (Fin d)) => v +ᵥ s)
```

lemma contDiffOn_vsub [source]

```
@[fun_prop]
lemma contDiffOn_vsub (s1 : Space d) :
    ContDiffOn ℝ ω (fun (s : Space d) => s -ᵥ s1) Set.univ
```

lemma vsub_differentiable [source]

```
@[fun_prop]
lemma vsub_differentiable {d} (s1 : Space d) :
    Differentiable ℝ (fun (s : Space d) => s -ᵥ s1)
```

lemma fderiv_space_components [source]

```
lemma fderiv_space_components {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (μ : Fin d) (f : M → Space d) (hf : Differentiable ℝ f) (m dm : M):
    fderiv ℝ f m dm μ  = fderiv ℝ (fun m' => f m' μ) m dm
```

def modelDiffeo [source]

A diffeomorphism between the two different manifold structures on `Space d`, that equivalent to `manifoldStructure d` and that equivalent to `𝓘(ℝ, Space d)`

```
noncomputable def modelDiffeo {d} :
    Diffeomorph (manifoldStructure d) 𝓘(ℝ, Space d) (Space d) (Space d) ⊤ where
```

lemma modelDiffeo_apply [source]

```
@[simp]
lemma modelDiffeo_apply (p : Space d) :
    modelDiffeo p = p
```

lemma basis_eq_mfderiv_modelDiffeo_single [source]

The derivative of `modelDiffeo` provides an equivalence between `Space d` and `EuclideanSpace ℝ (Fin d)`. This equivalences takes the basis of `EuclideanSpace ℝ (Fin d)` to the basis of `Space d`, and vice versa.

```
lemma basis_eq_mfderiv_modelDiffeo_single (d : ℕ) (μ : Fin d) (x : Space d) :
    basis μ = mfderiv (manifoldStructure d) 𝓘(ℝ, Space d) (modelDiffeo (d := d)) x
      (EuclideanSpace.single μ 1)
```

lemma norm_vadd_le_add [source]

```
lemma norm_vadd_le_add {d} (v : EuclideanSpace ℝ (Fin d)) (s : Space d) :
    ‖v +ᵥ s‖ ≤ ‖v‖ + ‖s‖
```

lemma differentiable_vadd [source]

```
@[fun_prop]
lemma differentiable_vadd {d} (v : EuclideanSpace ℝ (Fin d)) :
    Differentiable ℝ (fun (s : Space d) => v +ᵥ s)
```

lemma fderiv_vadd [source]

```
@[simp]
lemma fderiv_vadd {d} (v : EuclideanSpace ℝ (Fin d)) :
    fderiv ℝ (fun s => v +ᵥ s) = fun (_ : Space d) => ContinuousLinearMap.id ℝ _
```

lemma vadd_hasTemperateGrowth [source]

```
@[fun_prop]
lemma vadd_hasTemperateGrowth {d} (v : EuclideanSpace ℝ (Fin d)) :
    Function.HasTemperateGrowth (fun s : Space d => v +ᵥ s)
```

2.18 PhysLean.SpaceAndTime.Time.Derivatives

PhysLean/SpaceAndTime/Time/Derivatives.lean on GitHub.

lemma deriv_contDiff_of_space [source]

```
@[fun_prop]
lemma deriv_contDiff_of_space {n} {M : Type} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (f : Time → Space d → M) (hf : ContDiff ℝ (n + 1) ↿f) :
    ContDiff ℝ n fun (x : Space d) => (∂ₜ fun t => f t x) t
```

2.19 QuantumInfo.ClassicalInfo.Capacity

QuantumInfo/ClassicalInfo/Capacity.lean on GitHub.

structure Code [source]

*Here we define a *Code* by an encdoder and a decoder. The encoder is a function that takes strings (`Lists`) of any length over an alphabet `A`, and returns strings over `I`; the decoder takes `O` and gives back strings over `A`. The idea is that a channel would*

map from strings of `I` to `O`. Important special cases are where `I=O` (the channel doesn't change the symbol set), and `FixedLengthCode` where the output lengths only depend on input lengths.

```
structure Code where
  encoder : List A → List I
  decoder : List O → List A
```

structure FixedLengthCode [source]

A `FixedLengthCode` is a `Code` whose output lengths don't depend on the input content, only the input length.

```
structure FixedLengthCode extends Code A I O where
  enc_length : ℕ → ℕ
  enc_maps_length : ∀ as, (encoder as).length = enc_length (as.length)
  dec_length : ℕ → ℕ
  dec_maps_length : ∀ is, (decoder is).length = dec_length (is.length)
```

structure BlockCode [source]

A `BlockCode` is a `FixedLengthCode` that (1) maps symbols in discrete blocks of fixed length, (2) the encoded alphabet `I` has a canonical injection into `O`, and (3) has encoder and decoders that are inverses of each other. This is well-suited to describing noise and erasure channels. If the channel merely "corrupts" the data then this `O = I`; the erasure channel might for instance take `O = I ⊕ Unit` or `Option I`.

We define the behavior of a block code to "fail" if the input is not a multiple of the block size, by having it return an empty list.

```
structure BlockCode (io : I → O) extends FixedLengthCode A I O where
  block_in : ℕ
  block_out : ℕ
  block_enc : (Fin block_in → A) → (Fin block_out → I)
  block_dec : (Fin block_out → O) → (Fin block_in → A)
  block_enc_dec_inv : ∀ as, block_dec (io ∘ (block_enc as)) = as
  enc_length na := if na % block_in != 0 then 0 else (na / block_in) * block_out
  dec_length no := if no % block_out != 0 then 0 else (no / block_out) * block_in
  encoder as := if as.length % block_in != 0 then [] else
    sorry
  decoder os := if os.length % block_out != 0 then [] else
    sorry
  enc_maps_length := sorry
  dec_maps_length := sorry
```

2.20 QuantumInfo.ClassicalInfo.Channel

`QuantumInfo/ClassicalInfo/Channel.lean` on GitHub.

structure DMChannel [source]

Discrete Memoryless Channel. Each input symbol `I` has a corresponding output distribution `Distribution O`, and this process is applied independently on each symbol in the list.

```
structure DMChannel where
  symb_dist : I → Distribution O
```

def on_fin [source]

Apply a discrete memoryless channel to an n-character string.

```
def on_fin (C : DMChannel I O) {n : ℕ} (is : Fin n → I) : ProbDistribution (Fin n → O)
```

def on_list [source]

Apply a discrete memoryless channel to a list.

```
def on_list (C : DMChannel I O) (is : List I) : ProbDistribution (Fin (is.length) → O)
```

2.21 QuantumInfo.ClassicalInfo.Distribution

QuantumInfo/ClassicalInfo/Distribution.lean on GitHub.

def ProbDistribution [source]

We define our own (discrete) probability distribution notion here, instead of using `PMF` from Mathlib, because that uses ENNReals everywhere to maintain compatibility with `MeasureTheory.Measure`.

The probabilities internal to a Distribution are NNReals. This lets us more easily write the statement that they sum to 1, since NNReals can be added. (Probabilities, on their own, cannot.) But the FunLike instance gives `Prob` out, which carry the information that they are all in the range [0,1].

```
def ProbDistribution (α : Type u) [Fintype α] : Type u
```

def mk' [source]

Make a distribution, proving only that the values are nonnegative and that the sum is 1. The fact that the values are at most 1 is derived as a consequence.

```
def mk' (f : α → ℝ) (h₁ : ∀i, 0 ≤ f i) (hN : ∑ i, f i = 1) : ProbDistribution α
```

instance instFunLikeProb [source]

```
instance instFunLikeProb : FunLike (ProbDistribution α) α Prob where
```

theorem normalized [source]

```
@[simp]
theorem normalized (d : ProbDistribution α) : Finset.sum Finset.univ (fun i ↦ (d i : ℝ)) = 1
```

abbrev prob [source]

```
abbrev prob (d : ProbDistribution α)
```

theorem fun_eq_val [source]

```
@[simp]
theorem fun_eq_val (d : ProbDistribution α) : d.val = d
```

theorem funlike_apply [source]

```
@[simp]
theorem funlike_apply (d : α → Prob) (h : _) (x : α) :
    DFunLike.coe (self := instFunLikeProb) ⟨d, h⟩ x = d x
```

theorem ext [source]

```
@[ext]
theorem ext {p q : ProbDistribution α} (h : ∀ x, p x = q x) : p = q
```

theorem nonempty [source]

A distribution provides a witness that d is nonempty.

```
theorem nonempty (d : ProbDistribution α) : Nonempty α
```

def constant [source]

Make an constant distribution: supported on a single element. This is also called, variously, a "One-point distribution", a "Degenerate distribution", a "Deterministic distribution", a "Delta function", or a "Point mass distribution".

```
def constant (x : α) : ProbDistribution α
```

theorem constant_def [source]

```
theorem constant_def (x : α) : (constant x : α → Prob) = fun y ↦ if x = y then 1 else 0
```

theorem constant_eq [source]

```
@[simp]
theorem constant_eq (x : α) : constant x y = if x = y then 1 else 0
```

theorem constant_def' [source]

```
@[simp]
theorem constant_def' (x y : α) : (constant x : α → Prob) y = if x = y then 1 else 0
```

theorem constant_of_exists_one [source]

If a distribution has an element with probability 1, the distribution has a constant.

```
theorem constant_of_exists_one {D : ProbDistribution α} {x : α} (h : D x = 1) : D =
    ProbDistribution.constant x
```

def uniform [source]

Make an uniform distribution.

```
def uniform [n : Nonempty α] : ProbDistribution α
```

theorem uniform_def [source]

```
@[simp]
theorem uniform_def [Nonempty α] (y : α) : ((uniform y) : ℝ) = 1 / (Finset.univ.card (α := α))
```

def prod [source]

Make a distribution on a product of two Fintypes.

```
def prod (d1 : ProbDistribution α) (d2 : ProbDistribution β) : ProbDistribution (Prod α β)
```

theorem prod_def [source]

```
@[simp]
theorem prod_def (x : α) (y : β) : prod d1 d2 ⟨x, y⟩ = (d1 x) * (d2 y)
```

def extend_right [source]

Given a distribution on α, extend it to a distribution on `Sum α β` by giving it no support on `β`.

```
def extend_right (d : ProbDistribution α) : ProbDistribution (α ⊕ β)
```

def extend_left [source]

Given a distribution on α, extend it to a distribution on `Sum β α` by giving it no support on `β`.

```
def extend_left (d : ProbDistribution α) : ProbDistribution (β ⊕ α)
```

instance instMixable [source]

Make a convex mixture of two distributions on the same set.

```
instance instMixable : Mixable (α → ℝ) (ProbDistribution α)
```

def relabel [source]

Given a distribution on type α and an equivalence to type β, get the corresponding distribution on type β.

```
def relabel (d : ProbDistribution α) (σ : β ≃ α) : ProbDistribution β
```

def congr [source]

ProbDistribution on α and β are equivalent for equivalent types $\alpha \simeq \beta$.

```
def congr (σ : α ≃ β) : ProbDistribution α ≃ ProbDistribution β
```

theorem congr_apply [source]

```
@[simp]
theorem congr_apply (σ : α ≃ β) (d : ProbDistribution α) (j : β): (congr σ d) j = d (σ.symm j)
```

theorem congr_symm_apply [source]

The inverse and congruence operations for distributions commute

```
@[simp]
theorem congr_symm_apply (σ : α ≃ β) : (ProbDistribution.congr σ).symm = ProbDistribution.congr
    σ.symm
```

def coin [source]

The distribution on Fin 2 corresponding to a coin with probability p. Chance p of 1, 1-p of 0.

```
def coin (p : Prob) : ProbDistribution (Fin 2)
```

theorem coin_val_zero [source]

```
@[simp]
theorem coin_val_zero (p : Prob) : coin p 0 = p
```

theorem coin_val_one [source]

```
@[simp]
theorem coin_val_one (p : Prob) : coin p 1 = 1 - p
```

theorem fin_two_eq_coin [source]

Every distribution on two variable is some coin.

```
theorem fin_two_eq_coin (d : ProbDistribution (Fin 2)) : d = coin (d 0)
```

theorem coin_eq_iff [source]

```
theorem coin_eq_iff (p : Prob) (f : ProbDistribution (Fin 2)) :
    ProbDistribution.coin p = f ↔ p = f 0
```

structure RandVar [source]

A `T`-valued random variable over `α` is a map `var : α → T` along with a probability distribution `distr : Distribution α`.

```
structure RandVar (α : Type*) [Fintype α] (T : Type*) where
  var : α → T
  distr : ProbDistribution α
```

instance instFunctor [source]

```
instance instFunctor : Functor (RandVar α) where map f e
```

instance instLawfulFunctor [source]

```
instance instLawfulFunctor : LawfulFunctor (RandVar α) where
```

def expect_val [source]

`Distribution.exp_val` is the expectation value of a random variable `X`. Under the hood, it is the "convex combination over a finite family" on the type `T`, afforded by the `Mixable` instance, with the probability distribution of `X` as weights.

```
def expect_val (X : RandVar α T) : T
```

theorem expect_val_eq_mixable_mix [source]

The expectation value of a random variable over `α = Fin 2` is the same as `Mixable.mix` with probabiliy weight `X.distr 0`

```
theorem expect_val_eq_mixable_mix (d : ProbDistribution (Fin 2)) (x₁ x₂ : T) : expect_val (![x
    ₁, x₂], d) = Mixable.mix (d 0) x₁ x₂
```

theorem expect_val_constant [source]

The expectation value of a random variable with constant probability distribution `constant x` is its value at `x`

```
theorem expect_val_constant (x : α) (f : α → T) : expect_val (f, (constant x)) = f x
```

theorem zero_le_expect_val [source]

The expectation value of a nonnegative real random variable is also nonnegative

```
theorem zero_le_expect_val (d : ProbDistribution α) (f : α → ℝ) (hpos : 0 ≤ f) : 0 ≤ expect_val
    (f, d)
```

def congrRandVar [source]

`T`-valued random variables on `α` and `β` are equivalent if `α ≃ β`

```
def congrRandVar (σ : α ≃ β) : RandVar α T ≃ RandVar β T
```

def map_congr_eq_congr_map [source]

Given a `T`-valued random variable `X` over `α`, mapping over `T` commutes with the equivalence over `α`

```
def map_congr_eq_congr_map {S : Type _} [Mixable U S] (f : T → S) (σ : α ≃ β) (X : RandVar α T)
    :
  f <$> congrRandVar σ X = congrRandVar σ (f <$> X)
```

theorem expect_val_congr_eq_expect_val [source]

The expectation value is invariant under equivalence of random variables

```
@[simp]
theorem expect_val_congr_eq_expect_val (σ : α ≃ β) (X : RandVar α T) : expect_val (congrRandVar
    σ X) = expect_val X
```

2.22 QuantumInfo.ClassicalInfo.Entropy

QuantumInfo/ClassicalInfo/Entropy.lean on GitHub.

def H₁ [source]

*The one-event entropy function, $H_1(p)$ = -p*ln(p). Uses nits.*

```
def H₁ : Prob → ℝ
```

def H₁_zero_eq_zero [source]

H_1 of 0 is zero.

```
@[simp]
def H₁_zero_eq_zero : H₁ 0 = 0
```

def H₁_one_eq_zero [source]

H_1 of 1 is zero.

```
@[simp]
def H₁_one_eq_zero : H₁ 1 = 0
```

theorem H₁_nonneg [source]

Entropy is nonnegative.

```
theorem H₁_nonneg (p : Prob) : 0 ≤ H₁ p
```

theorem H₁_le_1 [source]

Entropy is less than 1.

```
theorem H₁_le_1 (p : Prob) : H₁ p < 1
```

theorem H₁_le_exp_m1 [source]

Entropy is at most 1/e.

```
theorem H₁_le_exp_m1 (p : Prob) : H₁ p ≤ Real.exp (-1)
```

theorem H₁_concave [source]

```
theorem H₁_concave : ∀ (x y : Prob), ∀ (p : Prob), p[H₁ x ↔ H₁ y] ≤ H₁ (p[x ↔ y])
```

def H_s [source]

The Shannon entropy of a discrete distribution, $H(X) = \sum H_1(p_x)$.

```
def Hₛ (d : ProbDistribution α) : ℝ
```

theorem H_s_nonneg [source]

Shannon entropy of a distribution is nonnegative.

```
theorem Hₛ_nonneg (d : ProbDistribution α) : 0 ≤ Hₛ d
```

theorem H_s_le_log_d [source]

Shannon entropy of a distribution is at most ln d.

```
theorem Hₛ_le_log_d (d : ProbDistribution α) : Hₛ d ≤ Real.log (Fintype.card α)
```

theorem H_s_constant_eq_zero [source]

The shannon entropy of a constant variable is zero.

```
@[simp]
theorem Hₛ_constant_eq_zero {i : α} : Hₛ (ProbDistribution.constant i) = 0
```

theorem H_s_uniform [source]

Shannon entropy of a uniform distribution is ln d.

```
theorem Hₛ_uniform [Nonempty α] :
    Hₛ (ProbDistribution.uniform (α := α)) = Real.log (Finset.univ.card (α := α))
```

theorem H_s_coin [source]

Shannon entropy of two-event distribution.

```
theorem Hₛ_coin (p : Prob) : Hₛ (ProbDistribution.coin p) = Real.binEntropy p
```

lemma H_s_eq_of_multiset_map_eq [source]

```
lemma Hₛ_eq_of_multiset_map_eq (d₁ : ProbDistribution α) (d₂ : ProbDistribution β)
    (h : Multiset.map d₁.prob Finset.univ.val = Multiset.map d₂.prob Finset.univ.val) :
    Hₛ d₁ = Hₛ d₂
```

2.23 QuantumInfo.ClassicalInfo.ForMathlib.Analysis.SpecialFunctions.Log.NegMulLog

QuantumInfo/ClassicalInfo/ForMathlib/Analysis/SpecialFunctions/Log/NegMulLog.lean on GitHub.

theorem negMulLog_strictMonoOn [source]

```
theorem negMulLog_strictMonoOn : StrictMonoOn Real.negMulLog (Set.Icc 0 (exp (-1)))
```

theorem negMulLog_strictAntiOn [source]

```
theorem negMulLog_strictAntiOn : StrictAntiOn Real.negMulLog (Set.Ici (exp (-1)))
```

theorem negMulLog_le_rexp_neg_one [source]

```
theorem negMulLog_le_rexp_neg_one {x : ℝ} (hx : 0 ≤ x) : negMulLog x ≤ exp (-1)
```

2.24 QuantumInfo.ClassicalInfo.Prob

QuantumInfo/ClassicalInfo/Prob.lean on GitHub.

def Prob [source]

Prob is a real number in the interval [0,1]. Similar to NNReal in many definitions, but this allows other nice things more 'safely' such as convex combination.

```
@[reducible]
def Prob
```

instance Coe Prob ℝ [source]

```
instance : Coe Prob ℝ
```

instance canLift [source]

```
instance canLift : CanLift ℝ Prob Subtype.val fun r => 0 ≤ r ∧ r ≤ 1
```

instance instZero [source]

```
instance instZero : Zero Prob
```

instance instOne [source]

```
instance instOne : One Prob
```

instance instMul [source]

```
instance instMul : Mul Prob
```

theorem coe_zero [source]

```
@[simp, norm_cast]
theorem coe_zero : (0 : Prob).val = 0
```

theorem coe_one [source]

```
@[simp, norm_cast]
theorem coe_one : (1 : Prob).val = 1
```

theorem coe_mul [source]

```
@[simp, norm_cast]
theorem coe_mul (x y : Prob) : (x * y).val = x.val * y.val
```

theorem coe_inf [source]

```
@[simp, norm_cast]
theorem coe_inf (x y : Prob) : (x ⊓ y).val = x.val ⊓ y.val
```

theorem coe_sup [source]

```
@[simp, norm_cast]
theorem coe_sup (x y : Prob) : (x ⊔ y).val = x.val ⊔ y.val
```

instance instCommMonoidWithZero [source]

```
instance instCommMonoidWithZero : CommMonoidWithZero Prob where
```

instance instDenselyOrdered [source]

```
instance instDenselyOrdered : DenselyOrdered Prob
```

instance instCompleteLinearOrder [source]

```
instance instCompleteLinearOrder : CompleteLinearOrder Prob
```

instance instInhabited [source]

```
instance instInhabited : Inhabited Prob where
```

instance LinearOrderedCommMonoidWithZero Prob [source]

```
instance : LinearOrderedCommMonoidWithZero Prob where
```

theorem zero_le_coe [source]

```
@[simp]
theorem zero_le_coe {p : Prob} : 0 ≤ (p : ℝ)
```

theorem coe_le_one [source]

```
@[simp]
theorem coe_le_one {p : Prob} : (p : ℝ) ≤ 1
```

theorem zero_le [source]

```
@[simp]
theorem zero_le {p : Prob} : 0 ≤ p
```

theorem le_one [source]

```
@[simp]
theorem le_one {p : Prob} : p ≤ 1
```

theorem ne_iff [source]

```
theorem ne_iff {x y : Prob} : (x : ℝ) ≠ (y : ℝ) ↔ x ≠ y
```

theorem toReal_mul [source]

```
@[simp, norm_cast]
theorem toReal_mul (x y : Prob) : (x * y : Prob) = (x : ℝ) * (y : ℝ)
```

theorem toNNReal_mk [source]

```
@[simp]
theorem toNNReal_mk : toNNReal { val := x, property := hx} = { val := x, property := hx.1 }
```

instance Coe Prob ℝ≥0 [source]

```
instance : Coe Prob ℝ≥0
```

instance canLiftNN [source]

```
instance canLiftNN : CanLift ℝ≥0 Prob toNNReal fun r => r ≤ 1
```

theorem eq_iff_nnreal [source]

```
protected theorem eq_iff_nnreal (n m : Prob) : (n : ℝ≥0) = (m : ℝ≥0) ↔ n = m
```

theorem toNNReal_zero [source]

```
@[simp, norm_cast]
theorem toNNReal_zero : (0 : Prob) = (0 : ℝ≥0)
```

theorem toNNReal_one [source]

```
@[simp, norm_cast]
theorem toNNReal_one : (1 : Prob) = (1 : ℝ≥0)
```

theorem ofNNReal_toNNReal [source]

```
theorem ofNNReal_toNNReal : ENNReal.ofNNReal (toNNReal p) = ENNReal.ofReal (p : ℝ)
```

def asProb [source]

```
def NNReal.asProb (p : ℝ≥0) (hp : p ≤ 1) : Prob
```

def asProb' [source]

```
def NNReal.asProb' (p : ℝ≥0) (hp : p.1 ≤ 1) : Prob
```

theorem zero_lt_coe [source]

```
theorem zero_lt_coe {p : Prob} (hp : p ≠ 0) : (0 : ℝ) < p
```

instance instSub [source]

Subtract a probability from another. Truncates to zero, so this is often not great to work with, for the same reason that Nat subtraction is a pain. But, it lets you write `1 - p`, which is sufficiently useful on its own that this seems worth having.

```
instance instSub : Sub Prob where
```

theorem coe_sub [source]

```
theorem coe_sub (p q : Prob) : (p - q : Prob)  = (p.val - q.val) ⊔ (0 : ℝ)
```

theorem coe_one_minus [source]

```
@[simp, norm_cast]
theorem coe_one_minus (p : Prob) : (1 - p : Prob) = 1 - (p : ℝ)
```

theorem add_one_minus [source]

```
theorem add_one_minus (p : Prob) : p.val + (1 - p).val = 1
```

theorem one_minus_inv [source]

```
@[simp]
theorem one_minus_inv (p : Prob) : 1 - (1 - p) = p
```

instance OrderTopology Prob [source]

```
instance : OrderTopology Prob
```

theorem coe_iInf [source]

```
@[simp, norm_cast]
theorem coe_iInf {ι : Type*} [Nonempty ι] (f : ι → Prob) : ↑(⨅ t, f t) = (⨅ t, f t : ℝ)
```

instance Nontrivial Prob [source]

```
instance : Nontrivial Prob where
```

theorem top_eq_one [source]

```
@[simp]
theorem top_eq_one : (⊤ : Prob) = 1
```

theorem sub_zero [source]

```
@[simp]
theorem sub_zero (p : Prob) : p - 0 = p
```

theorem toNNReal_Continuous [source]

```
@[fun_prop]
theorem toNNReal_Continuous : Continuous Prob.toNNReal
```

theorem mul_eq_one_iff [source]

```
@[simp]
theorem mul_eq_one_iff (p q : Prob) : p * q = 1 ↔ p = 1 ∧ q = 1
```

class Mixable [source]

A `Mixable T` typeclass instance gives a compact way of talking about the action of probabilities for forming linear combinations in convex spaces. The notation `p [ x₁ ↔ x₂ ]` means to take a convex combination, equal to `x₁` if `p=1` and to `x₂` if `p=0`.

Mixable is defined by an "underlying" data type `U` with addition and scalar multiplication, and a bijection between the `T` and a convex set of `U`. For instance, in `Mixable (Distribution (Fin n))`, `U` is `n`-element vectors (which form the probability simplex, degenerate in one dimension). For `QuantumInfo.Finite.MState` density matrices in quantum mechanics, which are PSD matrices of trace 1, `U` is the underlying matrix.

Why not just stick with existing notions of `Convex`? `Convex` requires that the type already forms an `AddCommMonoid` and `Module ℝ`. But many types, such as `Distribution`, are not: there is no good notion of "multiplying a probability distribution by 0.3" to get another distribution. We can coerce the distribution into, say, a vector or a function, but then we are not doing arithmetic with distributions. Accordingly, the expression `0.3 * distA + 0.7 * distB` cannot represent a distribution on its own.

```
class Mixable (U : outParam (Type u)) (T : Type v) [AddCommMonoid U] [Module ℝ U] where
  to_U : T → U
  to_U_inj : ∀ {T₁ T₂}, to_U T₁ = to_U T₂ → T₁ = T₂
  convex : Convex ℝ (Set.range to_U)
  mkT : {u : U} → (∃ t, to_U t = u) → { t : T // to_U t = u }
```

def mix_ab [source]

```
@[reducible]
def mix_ab [inst : Mixable U T] {a b : ℝ} (ha : 0 ≤ a) (hb : 0 ≤ b) (hab : a + b = 1) (x₁ x₂ :
    T) : T
```

def Mixable.mix [source]

`Mixable.mix` represents the notion of "convex combination" on the type `T`, afforded by the `Mixable` instance. It takes a `Prob`, that is, a `Real` between 0 and 1. For working directly with a Real, use `mix_ab`.

```
def mix [inst : Mixable U T] (p : Prob) (x₁ x₂ : T) : T
```

theorem to_U_of_mkT [source]

```
@[simp]
theorem to_U_of_mkT [inst : Mixable U T] (u : U) {h} : inst.to_U (mkT (u := u) h).1 = u
```

theorem mix_zero [source]

```
@[simp]
theorem mix_zero [inst : Mixable U T] (x₁ x₂ : T) : (0 : Prob) [ x₁ ↔ x₂ : inst] = x₂
```

instance instUniv [source]

When T is the whole space, and T is a suitable vector space over $\mathbb{R}$, we get a Mixable instance.

```
instance instUniv [AddCommMonoid T] [Module ℝ T] : Mixable T T where
```

theorem mkT_instUniv [source]

```
@[simp]
theorem mkT_instUniv [AddCommMonoid T] [Module ℝ T] {t : T} (h : ∃ t', to_U t' = t) : instUniv.
    mkT h = ⟨t, rfl⟩
```

theorem to_U_instUniv [source]

```
@[simp]
theorem to_U_instUniv [AddCommMonoid T] [Module ℝ T] {t : T} : instUniv.to_U t = t
```

theorem lem_1 [source]

```
theorem instPi.lem_1 {D : Type*} {T U : D → Type*} [∀i, AddCommMonoid (U i)] [∀ i, Module ℝ (U
    i)] [inst : ∀i, Mixable (U i) (T i)]
    {u : (i : D) → U i} (h : ∃ (t : (i : D) → T i), (fun d => to_U (t d)) = u) (d : D) : ∃ (t :
     T d), to_U t = u d
```

instance instPi [source]

Mixable instance on Pi types.

```
instance instPi : Mixable ((i:D) → U i) ((i:D) → T i) where
```

theorem val_mkT_instPi [source]

```
@[simp]
theorem val_mkT_instPi (D : Type*) [inst : Mixable U T] {u : D → U} (h : ∃ t, to_U t = u) : (
    instPi.mkT h).val =
    fun d ↦ (inst.mkT (instPi.lem_1 h d)).val
```

theorem to_U_instPi [source]

```
@[simp]
theorem to_U_instPi (D : Type*) [inst : Mixable U T] {t : D → T} : (instPi).to_U t = fun d ↦
    inst.to_U (t d)
```

def instSubtype [source]

Mixable instances on subtypes (of other mixable types), assuming that they have the correct closure properties.

```
def instSubtype {T : Type*} {P : T → Prop} (inst : Mixable U T)
    (h : ∀{x y:T},
      ∀{⦃a b : ℝ⦄}, (ha : 0 ≤ a) → (hb : 0 ≤ b) → (hab : a + b = 1) →
      P x → P y → P (inst.mix_ab ha hb hab x y))
    : Mixable U { t // P t} where
```

instance instMixable [source]

Probabilities `Prob` themselves are convex.

```
instance instMixable : Mixable ℝ Prob where
```

theorem to_U_mixable [source]

```
@[simp]
theorem to_U_mixable [AddCommMonoid T] [SMul ℝ T] (t : Prob) : instMixable.to_U t = t.val
```

theorem mkT_mixable [source]

```
@[simp]
theorem mkT_mixable (u : ℝ) (h : ∃ t : Prob, Mixable.to_U t = u) : Mixable.mkT h =
    ⟨u,Exists.casesOn h fun t ht ↦ ht ▸ t.2⟩, rfl⟩
```

abbrev Prob.mix [source]

`Prob.mix` is an alias of `Mixable.mix` so it can be accessed from a probability with dot notation, e.g. `p.mix x y`.

```
abbrev mix [AddCommMonoid U] [Module ℝ U] [inst : Mixable U T] (p : Prob) (x₁ x₂ : T)
```

def negLog [source]

Map a probability [0,1] to [0,+∞] with -log p. Special case that 0 maps to +∞ (not 0, as Real.log does). This makes it `Antitone`.

```
noncomputable def negLog : Prob → ENNReal
```

theorem negLog_Antitone [source]

```
theorem negLog_Antitone : Antitone negLog
```

theorem negLog_zero [source]

```
@[simp]
theorem negLog_zero : −log (0 : Prob) = ⊤
```

theorem negLog_one [source]

```
@[simp]
theorem negLog_one : −log 1 = 0
```

theorem negLog_eq_top_iff [source]

```
@[simp]
theorem negLog_eq_top_iff {p : Prob} : −log p = ⊤ ↔ p = 0
```

theorem negLog_pos_ENNReal [source]

```
theorem negLog_pos_ENNReal {p : Prob} (hp : p ≠ 0) : −log p = .ofNNReal ⟨-Real.log p,
    Left.nonneg_neg_iff.mpr (Real.log_nonpos p.2.1 p.2.2)⟩
```

theorem negLog_pos_Real [source]

```
@[simp]
theorem negLog_pos_Real {p : Prob} : (−log p).toReal = -Real.log p
```

theorem le_negLog_of_le_exp [source]

```
theorem le_negLog_of_le_exp {p : Prob} {x : ℝ} (h : p ≤ Real.exp (-x)) : ENNReal.ofReal x ≤ −
    log p
```

theorem negLog_ne_top [source]

```
@[aesop (rule_sets := [finiteness]) safe apply]
theorem negLog_ne_top {p : Prob} (hp : 0 < p.val) : −log p ≠ ∞
```

theorem negLog_eq_neg_ENNReal_log [source]

```
theorem negLog_eq_neg_ENNReal_log (p : Prob) : −log p = -ENNReal.log p
```

theorem negLog_eq_ofReal_neg_log [source]

```
theorem negLog_eq_ofReal_neg_log {p : Prob} (hp : 0 < p) :
    ENNReal.ofReal (-Real.log p) = −log p
```

theorem zero_lt_negLog [source]

```
@[simp]
theorem zero_lt_negLog {p : Prob} : 0 < -log p ↔ p ≠ 1
```

theorem Continuous_negLog [source]

```
@[fun_prop]
theorem Continuous_negLog : Continuous negLog
```

2.25 QuantumInfo.Finite.AxiomatizedEntropy.Defs

QuantumInfo/Finite/AxiomatizedEntropy/Defs.lean on GitHub.

class RelEntropy [source]

The axioms to be a well-behaved quantum relative entropy, as given by [Tomamichel](https://www.marcoto masterclass2022.pdf).

This simpler class allows for _trivial_ relative entropies, such as `-log tr(ρ⁰σ)`. Use mixing `RelEntropy.Nontrivial` to only allow nontrivial relative entropies.

```
class RelEntropy : Prop where
  DPI {d₁ d₂ : Type u} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq d₂]
    (ρ σ : MState d₁) (Λ : CPTPMap d₁ d₂) : f (Λ ρ) (Λ σ) ≤ f ρ σ
  of_kron {d₁ d₂ : Type u} [Fintype d₁] [Fintype d₂] [DecidableEq d₁] [DecidableEq d₂] :
    ∀ (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂), f (ρ₁ ⊗ ρ₂) (σ₁ ⊗ σ₂) = f ρ₁ σ₁ + f ρ₂ σ₂
  normalized {d : Type u} [fin : Fintype d] [DecidableEq d] [Nonempty d] (i : d) :
    f (.ofClassical (.constant i)) MState.uniform.M =
      some (Real.log fin.card, Real.log_nonneg (mod_cast Fintype.card_pos))
```

class Nontrivial [source]

The axioms to be a well-behaved quantum relative entropy, as given by [Tomamichel](https://www.marcoto masterclass2022.pdf).

```
class RelEntropy.Nontrivial [RelEntropy f] where
  nontrivial (d) [Fintype d] [DecidableEq d] : ∃ (ρ σ : MState d),
    ρ.M.support = ⊤ ∧ σ.M.support = ⊤ ∧ 0 < f ρ σ
```

theorem of_equiv_eq [source]

Relabelling a state with `CPTPMap.of_equiv` leaves relative entropies unchanged.

```
@[simp]
theorem of_equiv_eq (e : d ≃ d₂) (ρ σ : MState d) :
    f (CPTPMap.of_equiv e ρ) (CPTPMap.of_equiv e σ) = f ρ σ
```

theorem relabel_eq [source]

Relabelling a state with `MState.relabel` leaves relative entropies unchanged.

```
@[simp]
theorem relabel_eq (e : d₂ ≃ d) (ρ σ : MState d) :
    f (ρ.relabel e) (σ.relabel e) = f ρ σ
```

lemma wrt_self_eq_zero' [source]

The relative entropy is zero between any two states on a 1-D Hilbert space.

```
private lemma wrt_self_eq_zero' [Unique d] (ρ σ : MState d) : f ρ σ = 0
```

theorem wrt_self_eq_zero [source]

The relative entropy `D(ρ‖ρ) = 0`.

```
@[simp]
theorem wrt_self_eq_zero (ρ : MState d) : f ρ ρ.M = 0
```

theorem faithful [source]

*A nontrivial relative entropy is **faithful**, it can distinguish when two states are equal.*

```
theorem faithful (ρ σ : MState d) : f ρ σ = 0 ↔ ρ = σ
```

def min [source]

Quantum relative min-entropy.

```
def min (ρ : MState d) (σ : HermitianMat d ℂ) : ENNReal
```

theorem min_eq_top_iff [source]

```
@[aesop (rule_sets := [finiteness]) simp]
theorem min_eq_top_iff (ρ : MState d) (σ : HermitianMat d ℂ) :
    (min ρ σ) = ⊤ ↔ ρ.M.support ≤ σ.ker
```

theorem toReal_min [source]

```
protected theorem toReal_min (ρ : MState d) (σ : HermitianMat d ℂ) :
    (min ρ σ).toReal = -Real.log (ρ.exp_val {0 ≤ₚ σ})
```

instance RelEntropy min [source]

Min-relative entropy is a valid entropy function, albeit trivial (and not faithful).

```
instance : RelEntropy min where
```

theorem not_Nontrivial_min [source]

```
theorem not_Nontrivial_min : ¬Nontrivial min
```

theorem min_le [source]

The relative min-entropy is a lower bound on all relative entropies.

```
theorem min_le (ρ σ : MState d) : min ρ σ ≤ f ρ σ
```

def max [source]

Quantum relative max-entropy.

```
def max (ρ : MState d) (σ : HermitianMat d ℂ) : ENNReal
```

theorem max_not_top [source]

```
@[aesop (rule_sets := [finiteness]) simp]
protected theorem max_not_top (ρ : MState d) (σ : HermitianMat d ℂ) :
    (max ρ σ) ≠ ⊤ ↔ σ.ker ≤ ρ.M.ker
```

theorem toReal_max [source]

```
protected theorem toReal_max (ρ : MState d) (σ : HermitianMat d ℂ) :
    (max ρ σ).toReal = sInf { x : ℝ | ρ.M ≤ Real.exp x • σ }
```

theorem le_max [source]

The relative max-entropy is a lower bound on all relative entropies.

```
theorem le_max (ρ σ : MState d) : f ρ σ ≤ max ρ σ
```

class Entropy [source]

```
class Entropy (f : ∀ {d : Type u} [Fintype d] [DecidableEq d], MState d → ℝ≥0) where
  of_const {d : Type u} [Fintype d] [DecidableEq d] (ψ : Ket d) : f (.pure ψ) = 0
  of_kron {d₁ d₂ : Type u} [Fintype d₁] [Fintype d₂] [DecidableEq d₁] [DecidableEq d₂] :
    ∀ (ρ : MState d₁) (σ : MState d₂), f (ρ ⊗ σ) = f ρ + f σ
```

2.26 QuantumInfo.Finite.AxiomatizedEntropy.Renyi

QuantumInfo/Finite/AxiomatizedEntropy/Renyi.lean on GitHub.

def qRelativeEnt [source]

The quantum relative entropy $S(\rho\|\sigma) = Tr[\rho\ (log\ \rho - log\ \sigma)]$.

```
@[irreducible]
noncomputable def qRelativeEnt (ρ : MState d) (σ : HermitianMat d ℂ) : ENNReal
```

instance RelEntropy qRelativeEnt [source]

```
instance : RelEntropy qRelativeEnt where
```

instance RelEntropy.Nontrivial qRelativeEnt [source]

```
instance : RelEntropy.Nontrivial qRelativeEnt where
```

theorem qRelativeEnt_ker [source]

Quantum relative entropy as `Tr[ρ (log ρ - log σ)]` when supports are correct.

```
theorem qRelativeEnt_ker {ρ σ : MState d} (h : σ.M.ker ≤ ρ.M.ker) :
    (𝐃(ρ‖σ) : EReal) = ρ.M.inner (HermitianMat.log ρ - HermitianMat.log σ)
```

2.27 QuantumInfo.Finite.Braket

QuantumInfo/Finite/Braket.lean on GitHub.

structure Ket [source]

A ket as a vector of unit norm. We follow the convention in `Matrix` of vectors as simple functions from a Fintype. Kets are distinctly not a vector space in our notion, as they represent only normalized states and so cannot (in general) be added or scaled.

```
structure Ket where
  vec : d → ℂ
  normalized' : ∑ x, ‖vec x‖ ^ 2 = 1
```

structure Bra [source]

A bra is identical in definition to a `Ket`, but are separate to avoid complex conjugation confusion. They can be interconverted with the adjoint: `Ket.to_bra` and `Bra.to_ket`

```
structure Bra where
  vec : d → ℂ
  normalized' : ∑ x, ‖vec x‖ ^ 2 =1
```

instance instFunLikeKet [source]

```
instance instFunLikeKet : FunLike (Ket d) d ℂ where
```

instance instFunLikeBra [source]

```
instance instFunLikeBra : FunLike (Bra d) d ℂ where
```

def dot [source]

```
def dot (ξ : Bra d) (ψ : Ket d) : ℂ
```

theorem dot_eq_dotProduct [source]

```
theorem dot_eq_dotProduct (ψ : Bra d) (φ : Ket d) :⟨ψ‖φ⟩= dotProduct (m := d) ψ φ
```

theorem Ket.apply [source]

```
theorem Ket.apply (ψ : Ket d) (i : d) : ψ i = ψ.vec i
```

theorem Bra.apply [source]

```
theorem Bra.apply (ψ : Bra d) (i : d) : ψ i = ψ.vec i
```

theorem Ket.ext [source]

```
@[ext]
theorem Ket.ext {ξ ψ : Ket d} (h : ∀ x, ξ x = ψ x) : ξ = ψ
```

theorem Bra.ext [source]

```
@[ext]
theorem Bra.ext {ξ ψ : Bra d} (h : ∀ x, ξ x = ψ x) : ξ = ψ
```

theorem Ket.normalized [source]

```
theorem Ket.normalized (ψ : Ket d) : ∑ x, Complex.normSq (ψ x) = 1
```

theorem Bra.normalized [source]

```
theorem Bra.normalized (ψ : Bra d) : ∑ x, Complex.normSq (ψ x) = 1
```

def to_bra [source]

Any Bra can be turned into a Ket by conjugating the elements.

```
@[coe]
def Ket.to_bra (ψ : Ket d) : Bra d
```

def to_ket [source]

Any Ket can be turned into a Bra by conjugating the elements.

```
@[coe]
def Bra.to_ket (ψ : Bra d) : Ket d
```

instance instBraOfKet [source]

```
instance instBraOfKet : Coe (Ket d) (Bra d)
```

instance instKetOfBra [source]

```
instance instKetOfBra : Coe (Bra d) (Ket d)
```

theorem eq_conj [source]

```
@[simp]
theorem Bra.eq_conj (ψ : Ket d) (x : d) :⟨ψ| x = conj (|ψ⟩ x)
```

theorem apply' [source]

```
theorem Bra.apply' (ψ : Ket d) (i : d) : ⟨ψ| i = conj (ψ.vec i)
```

theorem Ket.exists_ne_zero [source]

```
theorem Ket.exists_ne_zero (ψ : Ket d) : ∃ x, ψ x ≠ 0
```

theorem Bra.exists_ne_zero [source]

```
theorem Bra.exists_ne_zero (ψ : Bra d) : ∃ x, ψ x ≠ 0
```

def Ket.normalize [source]

Create a ket out of a vector given it has a nonzero component

```
def Ket.normalize (v : d → ℂ) (h : ∃ x, v x ≠ 0) : Ket d
```

theorem Ket.normalize_ket_eq_self [source]

A ket is already normalized

```
theorem Ket.normalize_ket_eq_self (ψ : Ket d) : Ket.normalize (ψ.vec) (Ket.exists_ne_zero ψ) =
    ψ
```

def Bra.normalize [source]

Create a bra out of a vector given it has a nonzero component

```
def Bra.normalize (v : d → ℂ) (h : ∃ x, v x ≠ 0) : Bra d
```

def Bra.normalize_ket_eq_self [source]

A bra is already normalized

```
def Bra.normalize_ket_eq_self (ψ : Bra d) : Bra.normalize (ψ.vec) (Bra.exists_ne_zero ψ) = ψ
```

def uniform_superposition [source]

Ket form by the superposition of all elements in `d`. Commonly denoted by $|+\rangle$, especially for qubits

```
def uniform_superposition [hdne : Nonempty d] : Ket d
```

instance instInhabited [source]

There exists a ket for every nonempty `d`. Here, we use the uniform superposition

```
instance instInhabited [Nonempty d] : Inhabited (Ket d) where
```

def Ket.basis [source]

Construct the Ket corresponding to a basis vector, with a +1 phase.

```
def Ket.basis (i : d) : Ket d
```

def Bra.basis [source]

Construct the Bra corresponding to a basis vector, with a +1 phase.

```
def Bra.basis (i : d) : Bra d
```

instance instFunLikeBraket [source]

A Bra can be viewed as a function from Ket's to ℂ.

```
instance instFunLikeBraket : FunLike (Bra d) (Ket d) ℂ where
```

theorem dot_self_eq_one [source]

The inner product of any state with itself is 1.

```
@[simp]
theorem Braket.dot_self_eq_one (ψ : Ket d) :⟨ψ|ψ⟩= 1
```

def prod [source]

The outer product of two kets, creating an unentangled state.

```
def Ket.prod (ψ₁ : Ket d₁) (ψ₂ : Ket d₂) : Ket (d₁ × d₂) where
```

def IsProd [source]

A Ket is a product if it's `Ket.prod` of two kets.

```
def Ket.IsProd (ψ : Ket (d₁ × d₂)) : Prop
```

def IsEntangled [source]

A Ket is entangled if it's not `Ket.prod` of two kets.

```
def Ket.IsEntangled (ψ : Ket (d₁ × d₂)) : Prop
```

theorem IsProd_prod [source]

`Ket.prod` states are product states.

```
@[simp]
theorem Ket.IsProd_prod (ψ₁ : Ket d₁) (ψ₂ : Ket d₂) : (ψ₁.prod ψ₂).IsProd
```

theorem not_IsEntangled_prod [source]

`Ket.prod` states are not entangled states.

```
@[simp]
theorem Ket.not_IsEntangled_prod (ψ₁ : Ket d₁) (ψ₂ : Ket d₂) : ¬(ψ₁.prod ψ₂).IsEntangled
```

theorem IsProd_iff_mul_eq_mul [source]

A ket is a product state iff its components are cross-multiplicative.

```
theorem Ket.IsProd_iff_mul_eq_mul (ψ : Ket (d₁ × d₂)) : ψ.IsProd ↔
    ∀ i₁ i₂ j₁ j₂, ψ (i₁,j₁)  * ψ (i₂,j₂) = ψ (i₁,j₂) * ψ (i₂,j₁)
```

def MES [source]

The Maximally Entangled State, or MES, on a $d \times d$ system. In principle there are many, this is specifically the MES with an all-positive phase. For instance on `d := Fin 2`, this is the Bell state.

```
def Ket.MES (d) [Fintype d] [Nonempty d] : Ket (d × d) where
```

theorem MES_isEntangled [source]

On any space of dimension at least two, the maximally entangled state `MES` is entangled.

```
theorem Ket.MES_isEntangled [Nontrivial d] : (Ket.MES d).IsEntangled
```

theorem transposeTrick [source]

The transpose trick

```
theorem transposeTrick {d} [Fintype d] [Nonempty d] [DecidableEq d] {M : Matrix d d ℂ} :
    (M ⊗ₖ 1) *ᵥ (Ket.MES d).vec = (1 ⊗ₖ M.transpose) *ᵥ (Ket.MES d).vec
```

def PhaseEquiv [source]

The equivalence relation on `Ket` where two kets equivalent if they are equal up to a global phase, i.e. $\exists\, z, \|z\| = 1 \wedge a.vec = z \bullet b.vec$

```
def Ket.PhaseEquiv : Setoid (Ket d) where
```

def KetUpToPhase [source]

The type of `Kets` up to a global phase equivalence, as given by `Ket.PhaseEquiv`. In particular, `MStates` really only care about a KetUpToPhase, and not Kets themselves.

```
def KetUpToPhase
```

def mk [source]

Construct a `KetUpToPhase` from a `Ket`.

```
def KetUpToPhase.mk (ψ : Ket d) : KetUpToPhase d
```

def lift [source]

Lift a function on `Ket d` to `KetUpToPhase d`, given that it respects phase equivalence.

```
def KetUpToPhase.lift {α : Sort*} (f : Ket d → α)
    (hf : ∀ ψ φ, Ket.PhaseEquiv.r ψ φ → f ψ = f φ) : KetUpToPhase d → α
```

theorem lift_mk [source]

```
@[simp]
theorem KetUpToPhase.lift_mk {α : Sort*} (f : Ket d → α)
    (hf : ∀ ψ φ, Ket.PhaseEquiv.r ψ φ → f ψ = f φ) (ψ : Ket d) :
    KetUpToPhase.lift f hf (KetUpToPhase.mk ψ) = f ψ
```

theorem ind [source]

```
theorem KetUpToPhase.ind {p : KetUpToPhase d → Prop}
    (h : ∀ ψ : Ket d, p (KetUpToPhase.mk ψ)) : ∀ q, p q
```

theorem surjective_mk [source]

```
theorem KetUpToPhase.surjective_mk : Function.Surjective (KetUpToPhase.mk (d := d))
```

2.28 QuantumInfo.Finite.Capacity

QuantumInfo/Finite/Capacity.lean on GitHub.

def Emulates [source]

A channel Λ_1 `Emulates` another channel Λ_2 if there are D and E such that $D \circ \Lambda_1 \circ E = \Lambda_2$.

```
def Emulates (Λ₁ : CPTPMap d₁ d₂) (Λ₂ : CPTPMap d₃ d₄) : Prop
```

def εApproximates [source]

A channel A `εApproximates` channel B of the same dimensions if the for every state ρ, the fidelity $F(A(\rho), B(\rho))$ is at least 1-ε.

```
def εApproximates (A B : CPTPMap d₁ d₂) (ε : ℝ) : Prop
```

def AchievesRate [source]

A channel A `AchievesRate` R:$\mathbb{R}$ if for every ε>0, some n copies of A emulates a channel B such that log2(dimout(B))/$n \geq R$, and that B εApproximates the identity channel.

```
def AchievesRate (A : CPTPMap d₁ d₂) (R : ℝ) : Prop
```

def quantumCapacity [source]

```
noncomputable def quantumCapacity (A : CPTPMap d₁ d₂) : ℝ
```

theorem emulates_self [source]

Every quantum channel emulates itself.

```
@[refl]
theorem emulates_self (Λ : CPTPMap d₁ d₂) : Λ.Emulates Λ
```

theorem emulates_trans [source]

If a quantum channel A emulates B, and B emulates C, then A emulates C.

```
@[trans]
theorem emulates_trans (Λ₁ : CPTPMap d₁ d₂) (Λ₂ : CPTPMap d₃ d₄) (Λ₃ : CPTPMap d₅ d₆)
  (h₁₂ : Λ₁.Emulates Λ₂) (h₂₃ : Λ₂.Emulates Λ₃) : Λ₁.Emulates Λ₃
```

theorem εApproximates_self [source]

Every quantum channel perfectly approximates itself, that is, `εApproximates` with `ε = 0`.

```
theorem εApproximates_self (Λ : CPTPMap d₁ d₂) : Λ.εApproximates Λ 0
```

theorem εApproximates_monotone [source]

If a quantum channel A approximates B with ε_0, it also approximates B with all larger ε_1.

```
theorem εApproximates_monotone {A B : CPTPMap d₁ d₂} {ε₀ : ℝ} (h : A.εApproximates B ε₀)
    {ε₁ : ℝ} (h₂ : ε₀ ≤ ε₁) : A.εApproximates B ε₁
```

theorem achievesRate_0 [source]

Every quantum channel achieves a rate of zero.

```
theorem achievesRate_0 (Λ : CPTPMap d₁ d₂) : Λ.AchievesRate 0
```

theorem id_achievesRate_log_dim [source]

The identity channel on D dimensional space achieves a rate of log2(D).

```
theorem id_achievesRate_log_dim : (id (dIn := d₁)).AchievesRate (Real.logb 2 (Fintype.card d₁))
```

theorem not_achievesRate_gt_log_dim_in [source]

A channel cannot achieve a rate greater than log2(D), where D is the input dimension.

```
theorem not_achievesRate_gt_log_dim_in (Λ : CPTPMap d₁ d₂) {R : ℝ} (hR : Real.logb 2 (Fintype.
    card d₁) < R) :
    ¬Λ.AchievesRate R
```

theorem not_achievesRate_gt_log_dim_out [source]

A channel cannot achieve a rate greater than log2(D), where D is the output dimension.

```
theorem not_achievesRate_gt_log_dim_out (Λ : CPTPMap d₁ d₂) {R : ℝ} (hR : Real.logb 2 (Fintype.
    card d₂) < R): ¬Λ.AchievesRate R
```

theorem bddAbove_achievesRate [source]

The achievable rates are a bounded set.

```
theorem bddAbove_achievesRate (Λ : CPTPMap d₁ d₂) : BddAbove {R | Λ.AchievesRate R}
```

theorem zero_le_quantumCapacity [source]

Quantum channel capacity is nonnegative.

```
theorem zero_le_quantumCapacity (Λ : CPTPMap d₁ d₂) : 0 ≤ Λ.quantumCapacity
```

theorem quantumCapacity_ge_log_dim_in [source]

Quantum channel capacity is at most log2(D), where D is the input dimension.

```
theorem quantumCapacity_ge_log_dim_in (Λ : CPTPMap d₁ d₂) : Λ.quantumCapacity ≤ Real.logb 2 (
    Fintype.card d₁)
```

theorem coherentInfo_le_quantumCapacity [source]

The LSD (Lloyd-Shor-Devetak) theorem: the quantum capacity is at least as large the single-copy coherent information. The "coherent information" is used in literature to refer to both a function of state and a channel (`coherentInfo`), or a function of just a channel. In the latter case, the state is implicitly maximized over. Here we use the former definition and state that the lower bound is true for all states.

```
theorem coherentInfo_le_quantumCapacity (Λ : CPTPMap d₁ d₂) (ρ : MState d₁) : coherentInfo ρ Λ
    ≤ Λ.quantumCapacity
```

theorem quantumCapacity_eq_piProd_coherentInfo [source]

The quantum capacity is the limit of the coherent information of n-copy uses of the channel.

```
theorem quantumCapacity_eq_piProd_coherentInfo (Λ : CPTPMap d₁ d₂) : Λ.quantumCapacity =
    sSup { r : ℝ | ∃ n ρ, r = coherentInfo ρ (CPTPMap.piProd (fun (_ : Fin n) ↦ Λ))}
```

2.29 QuantumInfo.Finite.Channel.DegradableOrder

QuantumInfo/Finite/Channel/DegradableOrder.lean on GitHub.

def DegradablePreorder [source]

```
def DegradablePreorder (dIn : Type*) [Fintype dIn] [DecidableEq dIn] : Preorder
    (Σ dOut : (Σ t, Fintype t × DecidableEq t), let _ := dOut.snd.1; CPTPMap dIn dOut.fst)
    where
```

2.30 QuantumInfo.Finite.CPTPMap.Bundled

QuantumInfo/Finite/CPTPMap/Bundled.lean on GitHub.

structure HPMap [source]

Hermitian-preserving linear maps.

```
structure HPMap extends MatrixMap dIn dOut 𝕜 where
  HP : MatrixMap.IsHermitianPreserving toLinearMap
```

structure UnitalMap [source]

Unital linear maps.

```
structure UnitalMap [DecidableEq dIn] [DecidableEq dOut]
    extends MatrixMap dIn dOut R where
  unital : MatrixMap.Unital toLinearMap
```

structure TPMap [source]

Trace-preserving linear maps.

```
structure TPMap [Fintype dIn] [Fintype dOut] extends MatrixMap dIn dOut R where
  TP : MatrixMap.IsTracePreserving toLinearMap
```

structure PMap [source]

Positive linear maps.

```
structure PMap [Fintype dIn] [Fintype dOut]
    extends HPMap dIn dOut 𝕜 where
  pos : MatrixMap.IsPositive toLinearMap
  HP := pos.IsHermitianPreserving
```

structure CPMap [source]

Completely positive linear maps.

```
structure CPMap [Fintype dIn] [Fintype dOut] [DecidableEq dIn]
    extends PMap dIn dOut 𝕜 where
  cp : MatrixMap.IsCompletelyPositive toLinearMap
  pos := cp.IsPositive
```

structure PTPMap [source]

*Positive trace-preserving linear maps. These includes all channels, but aren't necessarily *completely* positive, see* `CPTPMap`*.*

```
structure PTPMap [Fintype dIn] [Fintype dOut]
  extends PMap dIn dOut 𝕜, TPMap dIn dOut 𝕜
```

structure PUMap [source]

Positive unital maps. These are important because they are the dual to `PTPMap`*: they are the most general way to map *observables*.*

```
structure PUMap [Fintype dIn] [Fintype dOut] [DecidableEq dIn] [DecidableEq dOut]
  extends PMap dIn dOut 𝕜, UnitalMap dIn dOut 𝕜
```

structure CPTPMap [source]

Completely positive trace-preserving linear maps. This is the most common meaning of "channel", often described as "the most general physically realizable quantum operation".

```
structure CPTPMap [Fintype dIn] [Fintype dOut] [DecidableEq dIn]
  extends PTPMap dIn dOut (𝕜 := 𝕜), CPMap dIn dOut 𝕜 where
```

structure CPUMap [source]

*Completely positive unital maps. These are important because they are the dual to `CPTPMap`: they are the physically realizable ways to map *observables*.*

```
structure CPUMap [Fintype dIn] [Fintype dOut] [DecidableEq dIn] [DecidableEq dOut]
  extends CPMap dIn dOut 𝕜, PUMap dIn dOut 𝕜
```

abbrev map [source]

```
abbrev map (M : HPMap dIn dOut 𝕜) : MatrixMap dIn dOut 𝕜
```

theorem HPMap.ext [source]

```
@[ext]
theorem ext (h : Λ₁.map = Λ₂.map) : Λ₁ = Λ₂
```

theorem funext_hermitian [source]

Two maps are equal if they agree on all Hermitian inputs.

```
theorem funext_hermitian (h : ∀ M : HermitianMat dIn ℂ, CΛ₁.map M = CΛ₂.map M) :
    CΛ₁ = CΛ₂
```

theorem funext_pos [source]

Two maps are equal if they agree on all positive inputs.

```
theorem funext_pos [Fintype dIn] (h : ∀ M : HermitianMat dIn ℂ, 0 ≤ M → CΛ₁.map M = CΛ₂.map M)
    :
    CΛ₁ = CΛ₂
```

theorem funext_pos_trace [source]

Two maps are equal if they agree on all positive inputs with trace one

```
theorem funext_pos_trace [Fintype dIn]
  (h : ∀ M : HermitianMat dIn ℂ, 0 ≤ M → M.trace = 1 → CΛ₁.map M = CΛ₂.map M) :
    CΛ₁ = CΛ₂
```

theorem funext_mstate [source]

Two maps are equal if they agree on all `MStates`.

```
theorem funext_mstate [Fintype dIn] [DecidableEq dIn] {Λ₁ Λ₂ : HPMap dIn dOut ℂ}
  (h : ∀ ρ : MState dIn, Λ₁.map ρ.m = Λ₂.map ρ.m) :
    Λ₁ = Λ₂
```

instance HPMap.instFunLike [source]

Hermitian-preserving maps are functions from `HermitianMats` to `HermitianMats`.

```
instance instFunLike : FunLike (HPMap dIn dOut ℂ) (HermitianMat dIn ℂ) (HermitianMat dOut ℂ)
    where
```

instance ContinuousLinearMapClass (HPMap dIn dOut ℂ) ℝ (HermitianMat dIn ℂ) (Her… [source]

```
instance [Fintype dIn] : ContinuousLinearMapClass
    (HPMap dIn dOut ℂ) ℝ (HermitianMat dIn ℂ) (HermitianMat dOut ℂ) where
```

theorem PMap.ext [source]

```
@[ext]
theorem ext {Λ₁ Λ₂ : PMap dIn dOut 𝕜} (h : Λ₁.map = Λ₂.map) : Λ₁ = Λ₂
```

theorem injective_toHPMap [source]

```
theorem injective_toHPMap : (PMap.toHPMap (dIn := dIn) (dOut := dOut) (𝕜 := 𝕜)).Injective
```

instance PMap.instFunLike [source]

Positive maps are functions from `HermitianMats` to `HermitianMats`.

```
instance instFunLike : FunLike (PMap dIn dOut ℂ) (HermitianMat dIn ℂ) (HermitianMat dOut ℂ)
    where
```

instance PMap.instLinearMapClass [source]

```
instance instLinearMapClass : LinearMapClass (PMap dIn dOut ℂ) ℝ (HermitianMat dIn ℂ) (
    HermitianMat dOut ℂ) where
```

instance instContinuousOrderHomClass [source]

```
instance instContinuousOrderHomClass : ContinuousOrderHomClass (PMap dIn dOut ℂ)
    (HermitianMat dIn ℂ) (HermitianMat dOut ℂ) where
```

theorem PMap.pos_Hermitian [source]

Positive-presering maps also preserve positivity on, specifically, Hermitian matrices.

```
@[simp]
theorem pos_Hermitian (M : PMap dIn dOut ℂ) {x : HermitianMat dIn ℂ} (h : 0 ≤ x) : 0 ≤ M x
```

def of_kraus_CPMap [source]

```
def of_kraus_CPMap {κ : Type*} [Fintype κ] [DecidableEq dIn] (M : κ → Matrix dOut dIn 𝕜) :
    CPMap dIn dOut 𝕜 where
```

theorem PTPMap.ext [source]

```
@[ext]
theorem ext {Λ₁ Λ₂ : PTPMap dIn dOut 𝕜} (h : Λ₁.map = Λ₂.map) : Λ₁ = Λ₂
```

theorem PTPMap.injective_toPMap [source]

```
theorem injective_toPMap : (PTPMap.toPMap (dIn := dIn) (dOut := dOut) (𝕜 := 𝕜)).Injective
```

instance PTPMap.instFunLike [source]

Positive trace-preserving maps are functions from `HermitianMats` to `HermitianMats`.

```
instance instFunLike : FunLike (PTPMap dIn dOut ℂ) (HermitianMat dIn ℂ) (HermitianMat dOut ℂ)
    where
```

instance PTPMap.instLinearMapClass [source]

```
instance instLinearMapClass : LinearMapClass (PTPMap dIn dOut ℂ) ℝ (HermitianMat dIn ℂ) (
    HermitianMat dOut ℂ) where
```

instance PTPMap.instHContinuousOrderHomClass [source]

```
instance instHContinuousOrderHomClass : ContinuousOrderHomClass (PTPMap dIn dOut ℂ)
    (HermitianMat dIn ℂ) (HermitianMat dOut ℂ) where
```

theorem PTPMap.pos_Hermitian [source]

PTP maps also preserve positivity on Hermitian matrices.

```
@[simp]
theorem pos_Hermitian (M : PTPMap dIn dOut ℂ) {x : HermitianMat dIn ℂ} (h : 0 ≤ x) : 0 ≤ M x
```

instance PTPMap.instMFunLike [source]

`PTPMaps` are functions from `MStates` to `MStates`.

```
instance instMFunLike [DecidableEq dIn] [DecidableEq dOut] :
    FunLike (PTPMap dIn dOut) (MState dIn) (MState dOut) where
```

instance instMContinuousMapClass [source]

```
instance instMContinuousMapClass [DecidableEq dIn] [DecidableEq dOut] :
    ContinuousMapClass (PTPMap dIn dOut) (MState dIn) (MState dOut) where
```

theorem val_apply_MState [source]

```
theorem val_apply_MState [DecidableEq dIn] (M : PTPMap dIn dOut) (ρ : MState dIn) :
    (M ρ : HermitianMat dOut ℂ) = (instFunLike.coe M) ρ
```

theorem nonemptyOut [source]

```
theorem nonemptyOut (Λ : PTPMap dIn dOut) [hIn : Nonempty dIn] [DecidableEq dIn] : Nonempty
    dOut
```

theorem CPTPMap.ext [source]

Two `CPTPMaps` are equal if their `MatrixMaps` are equal.

```
@[ext]
theorem ext {Λ₁ Λ₂ : CPTPMap dIn dOut 𝕜} (h : Λ₁.map = Λ₂.map) : Λ₁ = Λ₂
```

theorem injective_toPTPMap [source]

```
theorem injective_toPTPMap : (CPTPMap.toPTPMap (dIn := dIn) (dOut := dOut) (𝕜 := 𝕜)).Injective
```

instance CPTPMap.instMFunLike [source]

`CPTPMaps` are functions from `MStates` to `MStates`.

```
instance instMFunLike [DecidableEq dOut] : FunLike (CPTPMap dIn dOut) (MState dIn) (MState dOut
    ) where
```

theorem IsTracePreserving [source]

```
@[simp]
theorem IsTracePreserving (Λ : CPTPMap dIn dOut 𝕜) : Λ.map.IsTracePreserving
```

def of_kraus_CPTPMap [source]

```
def of_kraus_CPTPMap {κ : Type*} [Fintype κ] [DecidableEq dIn]
  (M : κ → Matrix dOut dIn 𝕜)
  (hTP : (∑ k, (M k).conjTranspose * (M k)) = 1) : CPTPMap dIn dOut 𝕜 where
```

theorem PUMap.ext [source]

```
@[ext]
theorem ext {Λ₁ Λ₂ : PUMap dIn dOut 𝕜} (h : Λ₁.map = Λ₂.map) : Λ₁ = Λ₂
```

theorem PUMap.injective_toPMap [source]

```
theorem injective_toPMap : (PUMap.toPMap (dIn := dIn) (dOut := dOut) (𝕜 := 𝕜)).Injective
```

instance PUMap.instFunLike [source]

`PUMaps` are functions from `HermitianMats` to `HermitianMats`.

```
instance instFunLike : FunLike (PUMap dIn dOut ℂ) (HermitianMat dIn ℂ) (HermitianMat dOut ℂ)
    where
```

instance PUMap.instLinearMapClass [source]

```
instance instLinearMapClass : LinearMapClass (PUMap dIn dOut ℂ) ℝ (HermitianMat dIn ℂ) (
    HermitianMat dOut ℂ) where
```

instance PUMap.instHContinuousOrderHomClass [source]

```
instance instHContinuousOrderHomClass : ContinuousOrderHomClass (PUMap dIn dOut ℂ)
    (HermitianMat dIn ℂ) (HermitianMat dOut ℂ) where
```

instance PUMap.instOneHomClass [source]

```
instance instOneHomClass : OneHomClass (PUMap dIn dOut ℂ)
    (HermitianMat dIn ℂ) (HermitianMat dOut ℂ) where
```

theorem PUMap.pos_Hermitian [source]

CPTP maps also preserve positivity on Hermitian matrices.

```
@[simp]
theorem pos_Hermitian (M : PUMap dIn dOut ℂ) {x : HermitianMat dIn ℂ} (h : 0 ≤ x) : 0 ≤ M x
```

theorem CPUMap.ext [source]

```
@[ext]
theorem ext {Λ₁ Λ₂ : CPUMap dIn dOut 𝕜} (h : Λ₁.map = Λ₂.map) : Λ₁ = Λ₂
```

theorem CPUMap.injective_toPMap [source]

```
theorem injective_toPMap : (CPMap.toPMap ∘ CPUMap.toCPMap (dIn := dIn) (dOut := dOut) (𝕜 := 𝕜))
    .Injective
```

instance CPUMap.instFunLike [source]

`CPUMaps` are functions from `HermitianMats` to `HermitianMats`.

```
instance instFunLike : FunLike (CPUMap dIn dOut ℂ) (HermitianMat dIn ℂ) (HermitianMat dOut ℂ)
    where
```

instance CPUMap.instLinearMapClass [source]

```
instance instLinearMapClass : LinearMapClass (CPUMap dIn dOut ℂ) ℝ (HermitianMat dIn ℂ) (
    HermitianMat dOut ℂ) where
```

instance CPUMap.instHContinuousOrderHomClass [source]

```
instance instHContinuousOrderHomClass : ContinuousOrderHomClass (CPUMap dIn dOut ℂ)
    (HermitianMat dIn ℂ) (HermitianMat dOut ℂ) where
```

instance CPUMap.instOneHomClass [source]

```
instance instOneHomClass : OneHomClass (CPUMap dIn dOut ℂ)
    (HermitianMat dIn ℂ) (HermitianMat dOut ℂ) where
```

theorem CPUMap.pos_Hermitian [source]

CPTP maps also preserve positivity on Hermitian matrices.

```
@[simp]
theorem pos_Hermitian (M : CPUMap dIn dOut ℂ) {x : HermitianMat dIn ℂ} (h : 0 ≤ x) : 0 ≤ M x
```

2.31 QuantumInfo.Finite.CPTPMap.CPTP

QuantumInfo/Finite/CPTPMap/CPTP.lean on GitHub.

def choi [source]

The Choi matrix of a CPTPMap.

```
@[reducible]
def choi
```

theorem choi_ext [source]

Two CPTPMaps are equal if their Choi matrices are equal.

```
theorem choi_ext {Λ₁ Λ₂ : CPTPMap dIn dOut} (h : Λ₁.choi = Λ₂.choi) : Λ₁ = Λ₂
```

theorem choi_PSD_of_CPTP [source]

The Choi matrix of a channel is PSD.

```
theorem choi_PSD_of_CPTP : Λ.map.choi_matrix.PosSemidef
```

theorem Tr_of_choi_of_CPTP [source]

The trace of a Choi matrix of a CPTP map is the cardinality of the input space.

```
@[simp]
theorem Tr_of_choi_of_CPTP : Λ.choi.trace =
    (Finset.univ (α := dIn)).card
```

def CPTP_of_choi_PSD_Tr [source]

Construct a CPTP map from a PSD Choi matrix with correct partial trace.

```
def CPTP_of_choi_PSD_Tr {M : Matrix (dOut × dIn) (dOut × dIn) ℂ} (h₁ : M.PosSemidef)
    (h₂ : M.traceLeft = 1) : CPTPMap dIn dOut where
```

theorem choi_of_CPTP_of_choi [source]

```
@[simp]
theorem choi_of_CPTP_of_choi (M : Matrix (dOut × dIn) (dOut × dIn) ℂ) {h₁} {h₂} :
    (CPTP_of_choi_PSD_Tr (M := M) h₁ h₂).choi = M
```

theorem mat_coe_eq_apply_mat [source]

```
theorem mat_coe_eq_apply_mat [DecidableEq dOut] (ρ : MState dIn) : (Λ ρ).m = Λ.map ρ.m
```

theorem funext [source]

```
@[ext]
theorem funext [DecidableEq dOut] {Λ₁ Λ₂ : CPTPMap dIn dOut} (h : ∀ ρ, Λ₁ ρ = Λ₂ ρ) : Λ₁ = Λ₂
```

def compose [source]

The composition of CPTPMaps, as a CPTPMap.

```
def compose (Λ₂ : CPTPMap dM dOut) (Λ₁ : CPTPMap dIn dM) : CPTPMap dIn dOut where
```

theorem compose_eq [source]

Composition of CPTPMaps by `CPTPMap.compose` is compatible with the `instFunLike` action.

```
@[simp]
theorem compose_eq [DecidableEq dOut] {Λ₁ : CPTPMap dIn dM} {Λ₂ : CPTPMap dM dOut} : ∀ρ, (Λ₂
    ∘ₘ Λ₁) ρ = Λ₂ (Λ₁ ρ)
```

theorem compose_assoc [source]

Composition of CPTPMaps is associative.

```
theorem compose_assoc [DecidableEq dOut] (Λ₃ : CPTPMap dM₂ dOut) (Λ₂ : CPTPMap dM dM₂)
    (Λ₁ : CPTPMap dIn dM) : (Λ₃ ∘ₘ Λ₂) ∘ₘ Λ₁ = Λ₃ ∘ₘ (Λ₂ ∘ₘ Λ₁)
```

instance instMixable [source]

CPTPMaps have a convex structure from their Choi matrices.

```
instance instMixable : Mixable (Matrix (dOut × dIn) (dOut × dIn) ℂ) (CPTPMap dIn dOut) where
```

def id [source]

The identity channel, which leaves the input unchanged.

```
def id : CPTPMap dIn dIn where
```

theorem id_map [source]

The map `CPTPMap.id` leaves any matrix unchanged.

```
@[simp]
theorem id_map : (id (dIn := dIn)).map = LinearMap.id
```

theorem id_MState [source]

The map `CPTPMap.id` leaves the input state unchanged.

```
@[simp]
theorem id_MState (ρ : MState dIn) : CPTPMap.id (dIn := dIn) ρ = ρ
```

theorem id_compose [source]

The map `CPTPMap.id` composed with any map is the same map.

```
@[simp]
theorem id_compose [DecidableEq dOut] (Λ : CPTPMap dIn dOut) : id ∘ₘ Λ = Λ
```

theorem compose_id [source]

Any map composed with `CPTPMap.id` is the same map.

```
@[simp]
theorem compose_id (Λ : CPTPMap dIn dOut) : Λ ∘ₘ id = Λ
```

def ofEquiv [source]

Given a equivalence (a bijection) between the types d_1 and d_2, that is, if they're the same dimension, then there's a CPTP channel for this. This is what we need for defining e.g. the SWAP channel, which is 'unitary' but takes heterogeneous input and outputs types $(d_1 \times d_2)$ and $(d_2 \times d_1)$.

```
def ofEquiv (σ : dIn ≃ dOut) : CPTPMap dIn dOut where
```

theorem ofEquiv_apply [source]

```
@[simp]
theorem ofEquiv_apply (σ : dIn ≃ dOut) (ρ : MState dIn) :
    ofEquiv σ ρ = ρ.relabel σ.symm
```

theorem equiv_inverse [source]

```
@[simp]
theorem equiv_inverse (σ : dIn ≃ dOut)  : (ofEquiv σ) ∘ (ofEquiv σ.symm) = id (dIn := dOut)
```

def SWAP [source]

The SWAP operation, as a channel.

```
def SWAP : CPTPMap (d₁ × d₂) (d₂ × d₁)
```

def assoc [source]

The associator, as a channel.

```
def assoc : CPTPMap ((d₁ × d₂) × d₃) (d₁ × d₂ × d₃)
```

def assoc' [source]

The inverse associator, as a channel.

```
def assoc' : CPTPMap (d₁ × d₂ × d₃) ((d₁ × d₂) × d₃)
```

theorem SWAP_eq_MState_SWAP [source]

```
@[simp]
theorem SWAP_eq_MState_SWAP (ρ : MState (d₁ × d₂)) : SWAP (d₁ := d₁) (d₂ := d₂) ρ = ρ.SWAP
```

theorem assoc_eq_MState_assoc [source]

```
@[simp]
theorem assoc_eq_MState_assoc (ρ : MState ((d₁ × d₂) × d₃)) : assoc (d₁ := d₁) (d₂ := d₂) (d₃
    := d₃) ρ = ρ.assoc
```

theorem assoc'_eq_MState_assoc' [source]

```
@[simp]
theorem assoc'_eq_MState_assoc' (ρ : MState (d₁ × d₂ × d₃)) : assoc' (d₁ := d₁) (d₂ := d₂) (d₃
    := d₃) ρ = ρ.assoc'
```

theorem assoc_assoc' [source]

```
@[simp]
theorem assoc_assoc' : (assoc (d₁ := d₁) (d₂ := d₂) (d₃ := d₃)) ∘ₘ assoc' = id
```

def traceLeft [source]

Partial tracing out the left, as a CPTP map.

```
@[simps]
def traceLeft : CPTPMap (d₁ × d₂) d₂
```

def traceRight [source]

Partial tracing out the right, as a CPTP map.

```
def traceRight : CPTPMap (d₁ × d₂) d₁
```

theorem traceLeft_eq_MState_traceLeft [source]

```
@[simp]
theorem traceLeft_eq_MState_traceLeft (ρ : MState (d₁ × d₂)) :
    traceLeft (d₁ := d₁) (d₂ := d₂) ρ = ρ.traceLeft
```

theorem traceRight_eq_MState_traceRight [source]

```
@[simp]
theorem traceRight_eq_MState_traceRight (ρ : MState (d₁ × d₂)) :
    traceRight (d₁ := d₁) (d₂ := d₂) ρ = ρ.traceRight
```

def replacement [source]

The replacement channel that maps all inputs to a given state.

```
def replacement [Nonempty dIn] [DecidableEq dOut] (ρ : MState dOut) : CPTPMap dIn dOut
```

theorem replacement_apply [source]

The output of `replacement ρ` is always that `ρ`.

```
@[simp]
theorem replacement_apply [Nonempty dIn] [DecidableEq dOut] (ρ : MState dOut) (ρ₀ : MState dIn)
    :
    replacement ρ ρ₀ = ρ
```

instance Inhabited (CPTPMap dIn dOut) [source]

```
instance [Nonempty dIn] [Nonempty dOut] [DecidableEq dOut] : Inhabited (CPTPMap dIn dOut)
```

instance Nonempty (CPTPMap dIn dOut) [source]

```
instance [Nonempty dIn] [Nonempty dOut] : Nonempty (CPTPMap dIn dOut)
```

def destroy [source]

There is a CPTP map that takes a system of any (nonzero) dimension and outputs the trivial Hilbert space, 1-dimensional, indexed by any `Unique` type. We can think of this as "destroying" the whole system; tracing out everything.

```
def destroy [Nonempty dIn] [Unique dOut] : CPTPMap dIn dOut
```

theorem eq_if_output_unique [source]

Two CPTP maps into the same one-dimensional output space must be equal

```
theorem eq_if_output_unique [Unique dOut] (Λ₁ Λ₂ : CPTPMap dIn dOut) : Λ₁ = Λ₂
```

instance instUnique [source]

There is exactly one CPTPMap to a 1-dimensional space.

```
instance instUnique [Nonempty dIn] [Unique dOut] : Unique (CPTPMap dIn dOut) where
```

theorem destroy_comp [source]

```
@[simp]
theorem destroy_comp {dOut₂ : Type*} [Unique dOut₂] [DecidableEq dOut] [Nonempty dIn] [Nonempty
    dOut]
  (Λ : CPTPMap dIn dOut) :
    destroy (dOut := dOut₂) ∘ₘ Λ = destroy
```

def prod [source]

The tensor product of two CPTPMaps.

```
def prod (Λ₁ : CPTPMap dI₁ dO₁) (Λ₂ : CPTPMap dI₂ dO₂) : CPTPMap (dI₁ × dI₂) (dO₁ × dO₂) where
```

def piProd [source]

Finitely-indexed tensor products of CPTPMaps.

```
def piProd (Λi : (i:ι) → CPTPMap (dI i) (dO i)) : CPTPMap ((i:ι) → dI i) ((i:ι) → dO i) where
```

theorem fin_1_piProd [source]

```
theorem fin_1_piProd
  {dI : Fin 1 → Type v} [Fintype (dI 0)] [DecidableEq (dI 0)]
  {dO : Fin 1 → Type w} [Fintype (dO 0)] [DecidableEq (dO 0)]
  (Λi : (i : Fin 1) → CPTPMap (dI 0) (dO 0)) :
    piProd Λi = sorry ∘ₘ ((Λi 1) ∘ₘ sorry)
```

theorem piProd_comp [source]

The tensor product of composed maps is the composition of the tensor products.

```
theorem piProd_comp
  {d₁ d₂ d₃ : ι → Type*}
  [∀ i, Fintype (d₁ i)] [∀ i, DecidableEq (d₁ i)]
  [∀ i, Fintype (d₂ i)] [∀ i, DecidableEq (d₂ i)]
  [∀ i, Fintype (d₃ i)] [∀ i, DecidableEq (d₃ i)]
  (Λ₁ : ∀ i, CPTPMap (d₁ i) (d₂ i)) (Λ₂ : ∀ i, CPTPMap (d₂ i) (d₃ i)) :
  piProd (fun i => (Λ₂ i) ∘ₘ (Λ₁ i)) = (piProd Λ₂) ∘ₘ (piProd Λ₁)
```

def ofUnitary [source]

Conjugating density matrices by a unitary as a channel. This is standard unitary evolution.

```
def ofUnitary (U : 𝐔[dIn]) : CPTPMap dIn dIn where
```

theorem ofUnitary_eq_conj [source]

The unitary channel U conjugated by U.

```
theorem ofUnitary_eq_conj (U : 𝐔[dIn]) (ρ : MState dIn) :
    (ofUnitary U) ρ = ρ.U_conj U
```

def IsUnitary [source]

A channel is unitary iff it is `ofUnitary U`.

```
def IsUnitary (Λ : CPTPMap dIn dIn) : Prop
```

theorem IsUnitary_iff_U_conj [source]

A channel is unitary iff it can be written as conjugation by a unitary.

```
theorem IsUnitary_iff_U_conj (Λ : CPTPMap dIn dIn) : IsUnitary Λ ↔ ∃ U, ∀ ρ, Λ ρ = ρ.U_conj U
```

theorem IsUnitary_equiv [source]

```
theorem IsUnitary_equiv (σ : dIn ≃ dIn) : IsUnitary (ofEquiv σ)
```

lemma kraus_sum_eq_one_of_TP [source]

```
private lemma kraus_sum_eq_one_of_TP
    {κ : Type*} [Fintype κ]
    {K₁ K₂ : κ → Matrix dOut dIn ℂ}
    (hTP : (MatrixMap.of_kraus K₁ K₂).IsTracePreserving) :
    ∑ k, (K₂ k).conjTranspose * (K₁ k) = 1
```

lemma exists_unitary_extending_isometry [source]

```
private lemma exists_unitary_extending_isometry
    {m n : Type*} [Fintype m] [Fintype n] [DecidableEq m] [DecidableEq n]
    (V : Matrix m n ℂ) (hV : V.conjTranspose * V = 1)
    (emb : n ↪ m) :
    ∃ U : 𝐔[m], ∀ i j, U.val i (emb j) = V i j
```

lemma purify_isometry_condition [source]

Given Kraus operators K indexed by (dOut × dIn), define the isometry matrix V : Matrix (dIn × dOut × dOut) dIn ℂ by V_{(a, b, d), a'} = (K (b, a))_{d, a'}. Then V†V = 1.

```
private lemma purify_isometry_condition
    {K : (dOut × dIn) → Matrix dOut dIn ℂ}
    (hTP : ∑ k, (K k).conjTranspose * (K k) = 1) :
    let V : Matrix (dIn × dOut × dOut) dIn ℂ
```

lemma purify_MState_pure_basis_default_entry [source]

```
private lemma purify_MState_pure_basis_default_entry (i j : dOut × dOut) :
    (MState.pure (Ket.basis (default : dOut × dOut))).m i j =
    if i = default ∧ j = default then 1 else 0
```

lemma purify_replacement_single_eq [source]

```
private lemma purify_replacement_single_eq (ρ₀ : MState (dOut × dOut)) (b₁ b₂ : dOut × dOut) :
    ((replacement ρ₀).map (Matrix.single () () 1)) b₁ b₂ = ρ₀.m b₁ b₂
```

lemma purify_prep_append_entry [source]

```
private lemma purify_prep_append_entry (X : Matrix dIn dIn ℂ)
      (a₁ : dIn) (b₁c₁ : dOut × dOut) (a₂ : dIn) (b₂c₂ : dOut × dOut) :
    let ρ₀
```

lemma purify_conj_entry [source]

```
private lemma purify_conj_entry (X : Matrix dIn dIn ℂ) (U : 𝐔[dIn × dOut × dOut])
      (i j : dIn × dOut × dOut) :
    let ρ₀
```

lemma purify_rhs_entry [source]

```
private lemma purify_rhs_entry (X : Matrix dIn dIn ℂ) (d₁ d₂ : dOut)
    (U : 𝐔[dIn × dOut × dOut]) :
    let ρ₀
```

lemma purify_of_kraus_entry [source]

```
private lemma purify_of_kraus_entry (K : (dOut × dIn) → Matrix dOut dIn ℂ) (X : Matrix dIn dIn
    ℂ) (d₁ d₂ : dOut) :
    (MatrixMap.of_kraus K K) X d₁ d₂ =
    ∑ k : dOut × dIn, ∑ α₁ : dIn, ∑ α₂ : dIn,
      (K k) d₁ α₁ * X α₁ α₂ * starRingEnd ℂ ((K k) d₂ α₂)
```

theorem exists_purify [source]

```
theorem exists_purify (Λ : CPTPMap dIn dOut) :
    ∃ (Λ' : CPTPMap (dIn × dOut × dOut) (dIn × dOut × dOut)),
      Λ'.IsUnitary ∧
      Λ = (
      let zero_prep : CPTPMap Unit (dOut × dOut) := replacement (MState.pure (Ket.basis default
    ))
      let prep := (id ⊗^cp zero_prep)
      let append : CPTPMap dIn (dIn × Unit) := CPTPMap.ofEquiv (Equiv.prodPUnit dIn).symm
      CPTPMap.traceLeft ∘ₘ CPTPMap.traceLeft ∘ₘ Λ' ∘ₘ prep ∘ₘ append
    )
```

def purify [source]

*Every channel can be written as a unitary channel on a larger system. In general, if the original channel was A→B, we may need to go as big as dilating the output system (the environment) by a factor of A*B. One way of stating this would be that it forms an isometry from A to (B×A×B). So that we can instead talk about the cleaner unitaries, we say that this is a unitary on (A×B×B). The defining properties that this is a valid purification comes are `purify_IsUnitary` and `purify_trace`. This means the environment always has type `dIn × dOut`.*

Furthermore, since we need a canonical "0" state on B in order to add with the input, we require a typeclass instance [Inhabited dOut].

```
def purify (Λ : CPTPMap dIn dOut) : CPTPMap (dIn × dOut × dOut) (dIn × dOut × dOut)
```

theorem purify_IsUnitary [source]

```
theorem purify_IsUnitary (Λ : CPTPMap dIn dOut) : Λ.purify.IsUnitary
```

theorem purify_trace [source]

*With a channel $\Lambda : A \to B$, a valid purification $(A\times B\times B)\to(A\times B\times B)$ is such that: * Preparing the default $|0\rangle$ state on two copies of B * Appending these to the input * Applying the purified unitary channel * Tracing out the two left parts of the output is equivalent to the original channel. This theorem states that the channel output by `purify` has this property.*

```
theorem purify_trace (Λ : CPTPMap dIn dOut) : Λ = (
    let zero_prep : CPTPMap Unit (dOut × dOut) := replacement (MState.pure (Ket.basis default))
    let prep := (id ⊗^cp zero_prep)
    let append : CPTPMap dIn (dIn × Unit) := CPTPMap.ofEquiv (Equiv.prodPUnit dIn).symm
    CPTPMap.traceLeft ∘ₘ CPTPMap.traceLeft ∘ₘ Λ.purify ∘ₘ prep ∘ₘ append
  )
```

def complementary [source]

The complementary channel comes from tracing out the other half (the right half) of the purified channel `purify`.

```
def complementary (Λ : CPTPMap dIn dOut) : CPTPMap dIn (dIn × dOut)
```

def IsDegradableTo [source]

*A channel is *degradable to* another, if the other can be written as a composition of a __degrading__ channel D with the original channel.*

```
def IsDegradableTo (Λ : CPTPMap dIn dOut) (Λ₂ : CPTPMap dIn dOut₂) : Prop
```

def IsAntidegradableTo [source]

*A channel is *antidegradable to* another, if the other `IsDegradableTo` this one.*

```
@[reducible]
def IsAntidegradableTo (Λ : CPTPMap dIn dOut) (Λ₂ : CPTPMap dIn dOut₂) : Prop
```

def IsDegradable [source]

*A channel is *degradable* if it `IsDegradableTo` its complementary channel.*

```
def IsDegradable (Λ : CPTPMap dIn dOut) : Prop
```

def IsAntidegradable [source]

*A channel is *antidegradable* if it `IsAntidegradableTo` its complementary channel.*

```
@[reducible]
def IsAntidegradable (Λ : CPTPMap dIn dOut) : Prop
```

instance instTop [source]

`CPTPMap`s inherit a topology from their choi matrices.

```
instance instTop : TopologicalSpace (CPTPMap dIn dOut)
```

theorem choi_IsEmbedding [source]

The projection from `CPTPMap` to the Choi matrix is an embedding

```
theorem choi_IsEmbedding : Topology.IsEmbedding (CPTPMap.choi (dIn := dIn) (dOut := dOut))
    where
```

instance instT3Space [source]

```
instance instT3Space : T3Space (CPTPMap dIn dOut)
```

2.32 QuantumInfo.Finite.CPTPMap.Dual

QuantumInfo/Finite/CPTPMap/Dual.lean on GitHub.

def MatrixMap.dual [source]

The dual of a map between matrices, defined by `Tr[A M(B)] = Tr[(dual M)(A) B]`. Sometimes called the adjoint of the map instead.

```
@[irreducible]
def dual (M : MatrixMap dIn dOut R) : MatrixMap dOut dIn R
```

theorem trace_eq [source]

The defining property of a dual map: inner products are preserved on the opposite argument.

```
theorem Dual.trace_eq (M : MatrixMap dIn dOut R) (A : Matrix dIn dIn R) (B : Matrix dOut dOut R
    ) :
    (M A * B).trace = (A * M.dual B).trace
```

theorem IsHermitianPreserving.dual [source]

The dual of a `IsHermitianPreserving` map also `IsHermitianPreserving`.

```
theorem IsHermitianPreserving.dual (h : M.IsHermitianPreserving) : M.dual.IsHermitianPreserving
```

theorem trace_mul_nonneg [source]

```
theorem _root_.Matrix.PosSemidef.trace_mul_nonneg {n : Type*} [Fintype n] [DecidableEq n]
    {A B : Matrix n n 𝕜} (hA : A.PosSemidef) (hB : B.PosSemidef) :
    0 ≤ (A * B).trace
```

theorem IsPositive.dual [source]

The dual of a `IsPositive` map also `IsPositive`.

```
theorem IsPositive.dual (h : M.IsPositive) : M.dual.IsPositive
```

theorem dual_Unital [source]

*The dual of TracePreserving map is *not* trace-preserving, it's *unital*, that is, M*(I) = I.*

```
theorem dual_Unital (h : M.IsTracePreserving) : M.dual.Unital
```

lemma dual_unique [source]

If two matrix maps satisfy the trace duality property, they are equal.

```
lemma dual_unique
    (M : MatrixMap dIn dOut 𝕜) (M' : MatrixMap dOut dIn 𝕜)
    (h : ∀ A B, (M A * B).trace = (A * M' B).trace) : M.dual = M'
```

lemma dual_choi_matrix [source]

The Choi matrix of the dual map is the transpose of the reindexed Choi matrix of the original map.

```
lemma dual_choi_matrix (M : MatrixMap dIn dOut 𝕜) :
    M.dual.choi_matrix = (M.choi_matrix.transpose).reindex (Equiv.prodComm dOut dIn) (Equiv.
    prodComm dOut dIn)
```

lemma dual_choi_matrix_posSemidef_of_posSemidef [source]

If the Choi matrix of a map is positive semidefinite, then the Choi matrix of its dual is also positive semidefinite.

```
lemma dual_choi_matrix_posSemidef_of_posSemidef (M : MatrixMap dIn dOut 𝕜) (h : M.choi_matrix.
    PosSemidef) :
    M.dual.choi_matrix.PosSemidef
```

lemma dual_id [source]

The dual of the identity map is the identity map.

```
lemma dual_id : (MatrixMap.id dIn 𝕜).dual = MatrixMap.id dIn 𝕜
```

lemma dual_kron [source]

The dual of a Kronecker product of maps is the Kronecker product of their duals.

```
lemma dual_kron {A B C D : Type*} [Fintype A] [Fintype B] [Fintype C] [Fintype D]
    [DecidableEq A] [DecidableEq B] [DecidableEq C] [DecidableEq D]
    (M : MatrixMap A B 𝕜) (N : MatrixMap C D 𝕜) :
    (M ⊗ₖₘ N).dual = M.dual ⊗ₖₘ N.dual
```

theorem IsCompletelyPositive.dual [source]

```
theorem IsCompletelyPositive.dual (h : M.IsCompletelyPositive) : M.dual.IsCompletelyPositive
```

lemma dualMap_toDualEquiv_symm_comp_toDualEquiv [source]

The composition of the dual of the inverse of the dual basis isomorphism with the dual basis isomorphism is the evaluation map.

```
lemma Module.Basis.dualMap_toDualEquiv_symm_comp_toDualEquiv {ι R M : Type*} [Fintype ι] [
    DecidableEq ι] [CommRing R] [AddCommGroup M] [Module R M] [Module.IsReflexive R M] (b :
    Module.Basis ι R M) :
    b.toDualEquiv.symm.toLinearMap.dualMap ∘ₗ b.toDualEquiv.toLinearMap = (Module.evalEquiv R M)
    .toLinearMap
```

lemma toDualEquiv_symm_comp_dualMap_toDualEquiv [source]

The composition of the inverse of the dual basis isomorphism with the dual of the dual basis isomorphism is the inverse of the evaluation map.

```
lemma Module.Basis.toDualEquiv_symm_comp_dualMap_toDualEquiv {ι R M : Type*} [Fintype ι] [
    DecidableEq ι] [CommRing R] [AddCommGroup M] [Module R M] [Module.IsReflexive R M] (b :
    Module.Basis ι R M) :
    b.toDualEquiv.symm.toLinearMap ∘ₗ b.toDualEquiv.toLinearMap.dualMap = (Module.evalEquiv R M)
    .symm.toLinearMap
```

theorem dual_dual [source]

```
@[simp]
theorem dual_dual : M.dual.dual = M
```

def CPTPMap.dual [source]

```
def dual (M : CPTPMap dIn dOut) : CPUMap dOut dIn where
```

theorem dual_pos [source]

```
theorem dual_pos (M : CPTPMap dIn dOut) {T : HermitianMat dOut ℂ} (hT : 0 ≤ T) :
    0 ≤ M.dual T
```

theorem dual.PTP_POVM [source]

The dual of a CPTP map preserves POVMs. Stated here just for two-element POVMs, that is, an operator `T` between 0 and 1.

```
theorem dual.PTP_POVM (M : CPTPMap dIn dOut) {T : HermitianMat dOut ℂ} (hT : 0 ≤ T ∧ T ≤ 1) :
    (0 ≤ M.dual T ∧ M.dual T ≤ 1)
```

theorem exp_val_Dual [source]

The defining property of a dual channel, as specialized to `MState.exp_val`.

```
theorem exp_val_Dual (ℰ : CPTPMap dIn dOut) (ρ : MState dIn) (T : HermitianMat dOut ℂ) :
    (ℰ ρ).exp_val T  = ρ.exp_val (ℰ.dual T)
```

def ofHermitianMat [source]

```
def HPMap.ofHermitianMat {dOut : Type*} (f : HermitianMat dIn ℂ →ₗ[ℝ] HermitianMat dOut ℂ) :
    HPMap dIn dOut where
```

theorem linearMap_ofHermitianMat [source]

```
@[simp]
theorem HPMap.linearMap_ofHermitianMat (f : HermitianMat dIn ℂ →ₗ[ℝ] HermitianMat dOut ℂ) :
    LinearMapClass.linearMap (HPMap.ofHermitianMat f) = f
```

theorem ofHermitianMat_linearMap [source]

```
@[simp]
theorem HPMap.ofHermitianMat_linearMap (f : HPMap dIn dOut ℂ) :
    ofHermitianMat (LinearMapClass.linearMap f) = f
```

def HPMap.hermDual [source]

```
def HPMap.hermDual : HPMap dOut dIn
```

theorem hermDual_hermDual [source]

```
@[simp]
theorem HPMap.hermDual_hermDual : f.hermDual.hermDual = f
```

theorem inner_hermDual [source]

The defining property of a dual map: inner products are preserved on the opposite argument.

```
theorem HPMap.inner_hermDual (B : HermitianMat dOut ℂ) :
    ⟪f A, B⟫ = ⟪A, f.hermDual B⟫
```

theorem inner_hermDual' [source]

Version of `HPMap.inner_hermDual` that uses HermitiaMat.inner directly. TODO cleanup

```
theorem HPMap.inner_hermDual' (B : HermitianMat dOut ℂ) :
    ⟪f A, B⟫ = ⟪A, f.hermDual B⟫
```

theorem IsPositive.hermDual [source]

The dual of a `IsPositive` map also `IsPositive`.

```
theorem MatrixMap.IsPositive.hermDual (h : MatrixMap.IsPositive f.map) : f.hermDual.map.
    IsPositive
```

theorem hermDual_Unital [source]

*The dual of TracePreserving map is *not* trace-preserving, it's *unital*, that is, M*(I) = I.*

```
theorem HPMap.hermDual_Unital [DecidableEq dIn] [DecidableEq dOut] (h : MatrixMap.
    IsTracePreserving f.map) :
    f.hermDual.map.Unital
```

def PTPMap.hermDual [source]

```
def hermDual (M : PTPMap dIn dOut) : PUMap dOut dIn where
```

theorem hermDual_pos [source]

```
theorem hermDual_pos (M : PTPMap dIn dOut) {T : HermitianMat dOut ℂ} (hT : 0 ≤ T) :
    0 ≤ M.hermDual T
```

theorem hermDual.PTP_POVM [source]

The dual of a PTP map preserves POVMs. Stated here just for two-element POVMs, that is, an operator `T` between 0 and 1.

```
theorem hermDual.PTP_POVM (M : PTPMap dIn dOut) {T : HermitianMat dOut ℂ} (hT : 0 ≤ T ∧ T ≤ 1)
    :
    (0 ≤ M.hermDual T ∧ M.hermDual T ≤ 1)
```

theorem exp_val_hermDual [source]

The defining property of a dual channel, as specialized to `MState.exp_val`.

```
theorem exp_val_hermDual (ℰ : PTPMap dIn dOut) (ρ : MState dIn) (T : HermitianMat dOut ℂ) :
    (ℰ ρ).exp_val T  = ρ.exp_val (ℰ.hermDual T)
```

2.33 QuantumInfo.Finite.CPTPMap.MatrixMap

QuantumInfo/Finite/CPTPMap/MatrixMap.lean on GitHub.

abbrev MatrixMap [source]

A `MatrixMap` is a linear map between squares matrices of size A to size B, over R.

```
abbrev MatrixMap (A B R : Type*) [Semiring R]
```

def id [source]

Alias of LinearMap.id, but specifically as a MatrixMap.

```
@[reducible]
def id : MatrixMap A A R
```

def choi_matrix [source]

Choi matrix of a given linear matrix map. Note that this is defined even for things that aren't CPTP, it's just rarely talked about in those contexts. This is the inverse of `MatrixMap.of_choi_matrix`. Compare with `MatrixMap.toMatrix`, which gives the transfer matrix.

```
def choi_matrix (M : MatrixMap A B R) : Matrix (B × A) (B × A) R
```

def of_choi_matrix [source]

Given the Choi matrix, generate the corresponding R-linear map between matrices as a MatrixMap. This is the inverse of `MatrixMap.choi_matrix`.

```
def of_choi_matrix (M : Matrix (B × A) (B × A) R) : MatrixMap A B R where
```

theorem map_choi_inv [source]

Proves that `MatrixMap.of_choi_matrix` *and* `MatrixMap.choi_matrix` *inverses.*

```
@[simp]
theorem map_choi_inv (M : Matrix (B × A) (B × A) R) : choi_matrix (of_choi_matrix M) = M
```

theorem choi_map_inv [source]

Proves that `MatrixMap.choi_matrix` *and* `MatrixMap.of_choi_matrix` *inverses.*

```
@[simp]
theorem choi_map_inv (M : MatrixMap A B R) : of_choi_matrix (choi_matrix M) = M
```

theorem choi_matrix_inj [source]

The correspondence induced by `MatrixMap.of_choi_matrix` *is injective.*

```
theorem choi_matrix_inj : Function.Injective (@choi_matrix A B R _ _)
```

def choi_equiv [source]

The linear equivalence between linear maps of matrices,and Choi matrices.

```
@[simps]
def choi_equiv : MatrixMap A B R ≃ₗ[R] Matrix (B × A) (B × A) R where
```

def toMatrix [source]

The linear equivalence between MatrixMap's and transfer matrices on a larger space. Compare with `MatrixMap.choi_matrix`*, which gives the Choi matrix instead of the transfer matrix.*

```
noncomputable def toMatrix [Fintype B] : MatrixMap A B R ≃ₗ[R] Matrix (B × B) (A × A) R
```

theorem toMatrix_comp [source]

Multiplication of transfer matrices, `MatrixMap.toMatrix`*, is equivalent to composition of maps.*

```
theorem toMatrix_comp [Fintype B] [Fintype C] [DecidableEq B] (M₁ : MatrixMap A B R) (M₂ :
    MatrixMap B C R) : toMatrix (M₂ ∘ₗ M₁) = (toMatrix M₂) * (toMatrix M₁)
```

def of_kraus [source]

Construct a matrix map out of families of matrices $M\ N : \Sigma \to$ *Matrix* $B\ A\ R$ *indexed by* κ *via* $X \mapsto \sum k : \kappa$, $(M\ k) * X * (N\ k)^H$

```
def of_kraus (M N : κ → Matrix B A R) : MatrixMap A B R
```

def exists_kraus [source]

```
def exists_kraus (Φ : MatrixMap A B R) : ∃ r : ℕ, ∃ (M N : Fin r → Matrix B A R), Φ = of_kraus
    M N
```

def submatrix [source]

The `MatrixMap` corresponding to applying a `submatrix` operation on each side.

```
@[simps]
def submatrix (f : B → A) : MatrixMap A B R where
```

theorem submatrix_id [source]

```
@[simp]
theorem submatrix_id : submatrix R _root_.id = id A R
```

theorem submatrix_comp [source]

```
@[simp]
theorem submatrix_comp (f : C → B) (g : B → A) :
    submatrix R f ∘ₗ submatrix R g = submatrix R (g ∘ f)
```

def kron [source]

The Kronecker product of MatrixMaps. Defined here using `TensorProduct.map` M_1 M_2, with appropriate reindexing operations and `LinearMap.toMatrix`/`Matrix.toLin`. Notation $\otimes_{k\,m}$.

```
noncomputable def kron [CommSemiring R] (M₁ : MatrixMap A B R) (M₂ : MatrixMap C D R) :
    MatrixMap (A × C) (B × D) R
```

theorem kron_def [source]

The extensional definition of the Kronecker product `MatrixMap.kron`, in terms of the entries of its image.

```
theorem kron_def [CommSemiring R] (M₁ : MatrixMap A B R) (M₂ : MatrixMap C D R) (M : Matrix (A
    × C) (A × C) R) :
    (M₁ ⊗ₖ M₂) M (b₁, d₁) (b₂, d₂) = ∑ a₁, ∑ a₂, ∑ c₁, ∑ c₂,
      (M₁ (Matrix.single a₁ a₂ 1) b₁ b₂) * (M₂ (Matrix.single c₁ c₂ 1) d₁ d₂) * (M (a₁, c₁) (a
    ₂, c₂))
```

theorem add_kron [source]

```
theorem add_kron (ML₁ ML₂ : MatrixMap A B R) (MR : MatrixMap C D R) : (ML₁ + ML₂) ⊗ₖ MR = ML₁
    ⊗ₖ MR + ML₂ ⊗ₖ MR
```

theorem kron_add [source]

```
theorem kron_add (ML : MatrixMap A B R) (MR₁ MR₂ : MatrixMap C D R) : ML ⊗ₖₘ (MR₁ + MR₂) = ML
    ⊗ₖₘ MR₁ + ML ⊗ₖₘ  MR₂
```

theorem smul_kron [source]

```
theorem smul_kron (r : R) (ML : MatrixMap A B R) (MR : MatrixMap C D R) : (r • ML) ⊗ₖₘ MR = r
    • (ML ⊗ₖₘ MR)
```

theorem kron_smul [source]

```
theorem kron_smul (r : R) (ML : MatrixMap A B R) (MR : MatrixMap C D R) : ML ⊗ₖₘ (r • MR) = r
    • (ML ⊗ₖₘ MR)
```

theorem zero_kron [source]

```
@[simp]
theorem zero_kron (MR : MatrixMap C D R) : (0 : MatrixMap A B R) ⊗ₖₘ MR = 0
```

theorem kron_zero [source]

```
@[simp]
theorem kron_zero (ML : MatrixMap A B R) : ML ⊗ₖₘ (0 : MatrixMap C D R) = 0
```

theorem kron_id_id [source]

```
theorem kron_id_id : (id A R ⊗ₖₘ id B R) = id (A × B) R
```

theorem kron_comp_distrib [source]

For maps L_1, L_2, R_1, and R_2, the product $(L_2 \circ_l L_1) \otimes_{km} (R_2 \circ_l R_1) = (L_2 \otimes_{km} R_2) \circ_l (L_1 \otimes_{km} R_1)$

```
theorem kron_comp_distrib (L₁ : MatrixMap Dl₁ Dl₂ R) (L₂ : MatrixMap Dl₂ Dl₃ R) (R₁ : MatrixMap
    Dr₁ Dr₂ R)
   (R₂ : MatrixMap Dr₂ Dr₃ R) : (L₂ ∘ₗ L₁) ⊗ₖₘ (R₂ ∘ₗ R₁) = (L₂ ⊗ₖₘ R₂) ∘ₗ (L₁ ⊗ₖₘ R₁)
```

theorem kron_map_of_kron_state [source]

The operational definition of the Kronecker product `MatrixMap.kron`, that it maps a Kronecker product of inputs to the Kronecker product of outputs. It is the unique bilinear map doing so.

```
theorem kron_map_of_kron_state [CommRing R] (M₁ : MatrixMap A B R) (M₂ : MatrixMap C D R) (MA :
    Matrix A A R) (MC : Matrix C C R) : (M₁ ⊗ₖₘ M₂) (MA ⊗ₖ MC) = (M₁ MA) ⊗ₖ (M₂ MC)
```

theorem choi_matrix_state_rep [source]

```
theorem choi_matrix_state_rep {B : Type*} [Fintype B] [Nonempty A] (M : MatrixMap A B ℂ) :
   M.choi_matrix = (↑(Fintype.card (α := A)) : ℂ) • (M ⊗ₖₘ (LinearMap.id : MatrixMap A A ℂ)) (
   MState.pure (Ket.MES A)).m
```

theorem submatrix_kron_submatrix [source]

```
theorem submatrix_kron_submatrix [CommSemiring R] (f : B → A) (g : D → C) :
    submatrix R f ⊗ₖₘ submatrix R g = submatrix R (Prod.map f g)
```

theorem submatrix_kron_id [source]

```
theorem submatrix_kron_id [CommSemiring R] (f : B → A) :
    submatrix R f ⊗ₖₘ id C R = submatrix R (Prod.map f _root_.id)
```

theorem id_kron_submatrix [source]

```
theorem id_kron_submatrix [CommSemiring R] (f : B → A) :
    id C R ⊗ₖₘ submatrix R f = submatrix R (Prod.map _root_.id f)
```

opaque piTensorProduct [source]

Like `Basis.tensorProduct`, but for `PiTensorProduct`

```
noncomputable opaque _root_.Module.Basis.piTensorProduct [∀i, Fintype (L i)]
    (b : (i:ι) → Module.Basis (L i) R (s i)) :
      Module.Basis ((i:ι) → L i) R (PiTensorProduct R s)
```

def piProd [source]

Finite Pi-type tensor product of MatrixMaps. Defined as `PiTensorProduct.tprod` of the underlying Linear maps. Notation ⨂ₜₘ`[R] i, f i`, eventually.

```
noncomputable def piProd (Λi : ∀ i, MatrixMap (dI i) (dO i) R) : MatrixMap (∀i, dI i) (∀i, dO i
    ) R
```

theorem piProd_comp [source]

Composition of `MatrixMap.piProd` maps distributes over the tensor product.

```
theorem piProd_comp
  {d₁ d₂ d₃ : ι → Type*}
  [∀ i, Fintype (d₁ i)] [∀ i, DecidableEq (d₁ i)]
  [∀ i, Fintype (d₂ i)] [∀ i, DecidableEq (d₂ i)]
  [∀ i, Fintype (d₃ i)] [∀ i, DecidableEq (d₃ i)]
  (Λ₁ : ∀ i, MatrixMap (d₁ i) (d₂ i) R) (Λ₂ : ∀ i, MatrixMap (d₂ i) (d₃ i) R) :
    piProd (fun i ↦ (Λ₂ i) ∘ₗ (Λ₁ i)) = (piProd Λ₂) ∘ₗ (piProd Λ₁)
```

2.34 QuantumInfo.Finite.CPTPMap.Unbundled

QuantumInfo/Finite/CPTPMap/Unbundled.lean on GitHub.

def IsTracePreserving [source]

*A linear matrix map is *trace preserving* if trace of the output equals trace of the input.*

```
def IsTracePreserving (M : MatrixMap A B R) : Prop
```

theorem IsTracePreserving_iff_trace_choi [source]

A map is trace preserving iff the partial trace of the Choi matrix is the identity.

```
theorem IsTracePreserving_iff_trace_choi [DecidableEq A] (M : MatrixMap A B R) : M.
    IsTracePreserving
    ↔ M.choi_matrix.traceLeft = 1
```

theorem apply_trace [source]

Simp lemma: the trace of the image of a IsTracePreserving map is the same as the original trace.

```
@[simp]
theorem apply_trace (h : M.IsTracePreserving) (ρ : Matrix A A R)
    : (M ρ).trace = ρ.trace
```

theorem trace_choi [source]

The trace of a Choi matrix of a TP map is the cardinality of the input space.

```
theorem trace_choi [DecidableEq A] (h : M.IsTracePreserving) :
    M.choi_matrix.trace = (Finset.univ (α := A)).card
```

theorem IsTracePreserving.comp [source]

The composition of IsTracePreserving maps is also trace preserving.

```
theorem comp (h₁ : M.IsTracePreserving) (h₂ : M₂.IsTracePreserving) :
    IsTracePreserving (M₂ ∘ₗ M)
```

theorem IsTracePreserving.id [source]

The identity MatrixMap IsTracePreserving.

```
@[simp]
theorem id : (id A R).IsTracePreserving
```

theorem unit_linear [source]

Unit linear combinations of IsTracePreserving maps are IsTracePreserving.

```
theorem unit_linear {M₁ M₂ : MatrixMap A B R} {x y : R}
    (h₁ : M₁.IsTracePreserving) (h₂ : M₂.IsTracePreserving) (hxy : x + y = 1) :
    (x • M₁ + y • M₂).IsTracePreserving
```

theorem IsTracePreserving.kron [source]

The kronecker product of IsTracePreserving maps is also trace preserving.

```
theorem kron {M₁ : MatrixMap A B R} {M₂ : MatrixMap C D R} (h₁ : M₁.IsTracePreserving) (h₂ : M₂.
    IsTracePreserving) :
    (M₁ ⊗ₖₘ M₂).IsTracePreserving
```

theorem of_kraus_isTracePreserving [source]

The channel $X \mapsto \sum k : \kappa, (M\ k) * X * (N\ k)^H$ formed by Kraus operators M, $N : \kappa \to$ Matrix B A R is trace-preserving if $\sum k : \kappa, (N\ k)^H * (M\ k) = 1$

```
theorem of_kraus_isTracePreserving
  (M N : κ → Matrix B A S)
  (hTP : (∑ k, (N k).conjTranspose * (M k)) = 1) :
  (MatrixMap.of_kraus M N).IsTracePreserving
```

theorem IsTracePreserving.submatrix [source]

`MatrixMap.submatrix` is trace-preserving when the function is an equivalence.

```
theorem submatrix (e : A ≃ B) : (MatrixMap.submatrix R e).IsTracePreserving
```

def Unital [source]

*A linear matrix map is *unital* if it preserves the identity.*

```
def Unital (M : MatrixMap A B R) : Prop
```

theorem map_1 [source]

```
@[simp]
theorem map_1 (h : M.Unital) : M 1 = 1
```

theorem Unital.id [source]

The identity `MatrixMap` is `Unital`.

```
@[simp]
theorem id : (id A R).Unital
```

def MatrixMap.IsHermitianPreserving [source]

*A linear matrix map is *Hermitian preserving* if it maps `IsHermitian` matrices to `IsHermitian`.*

```
def IsHermitianPreserving (M : MatrixMap A B R) : Prop
```

def MatrixMap.IsPositive [source]

*A linear matrix map is *positive* if it maps `PosSemidef` matrices to `PosSemidef`.*

```
def IsPositive [Fintype A] [Fintype B] (M : MatrixMap A B R) : Prop
```

def IsCompletelyPositive [source]

*A linear matrix map is *completely positive* if, for any integer n, the tensor product with `I(n)` is positive.*

```
def IsCompletelyPositive [Fintype A] [Fintype B] [DecidableEq A] (M : MatrixMap A B R) : Prop
```

theorem IsHermitianPreserving.id [source]

The identity MatrixMap IsHermitianPreserving.

```
theorem id : (id A R).IsPositive
```

theorem IsHermitianPreserving.comp [source]

The composition of IsHermitianPreserving maps is also Hermitian preserving.

```
theorem comp {M₁ : MatrixMap A B R} {M₂ : MatrixMap B C R}
    (h₁ : M₁.IsHermitianPreserving) (h₂ : M₂.IsHermitianPreserving) : IsHermitianPreserving (M
    ₂ ∘ₗ M₁)
```

theorem IsPositive.IsHermitianPreserving [source]

```
theorem IsHermitianPreserving {M : MatrixMap A B R}
    (hM : IsPositive M) : IsHermitianPreserving M
```

theorem IsPositive.comp [source]

The composition of IsPositive maps is also positive.

```
theorem comp {M₁ : MatrixMap A B R} {M₂ : MatrixMap B C R} (h₁ : M₁.IsPositive)
    (h₂ : M₂.IsPositive) : IsPositive (M₂ ∘ₗ M₁)
```

theorem IsPositive.id [source]

The identity MatrixMap IsPositive.

```
@[simp]
theorem id : (id A R).IsPositive
```

theorem IsPositive.add [source]

Sums of IsPositive maps are IsPositive.

```
theorem add {M₁ M₂ : MatrixMap A B R} (h₁ : M₁.IsPositive) (h₂ : M₂.IsPositive) :
    (M₁ + M₂).IsPositive
```

theorem IsPositive.smul [source]

Nonnegative scalings of IsPositive maps are IsPositive.

```
theorem smul {M : MatrixMap A B R} (hM : M.IsPositive) {x : R} (hx : 0 ≤ x) :
    (x • M).IsPositive
```

theorem of_Fintype [source]

Definition of a CP map, but with `Fintype T` in the definition instead of a `Fin n`.

```
theorem of_Fintype  {M : MatrixMap A B R} (h : IsCompletelyPositive M)
    (T : Type*) [Fintype T] [DecidableEq T] :
    (M.kron (LinearMap.id : MatrixMap T T R)).IsPositive
```

theorem IsCompletelyPositive.IsPositive [source]

```
theorem IsPositive {M : MatrixMap A B R}
    (hM : IsCompletelyPositive M) : IsPositive M
```

theorem IsCompletelyPositive.comp [source]

The composition of IsCompletelyPositive maps is also completely positive.

```
theorem comp [DecidableEq B] {M₁ : MatrixMap A B R} {M₂ : MatrixMap B C R} (h₁ : M₁.
    IsCompletelyPositive)
    (h₂ : M₂.IsCompletelyPositive) : IsCompletelyPositive (M₂ ∘ₗ M₁)
```

theorem IsCompletelyPositive.id [source]

The identity MatrixMap IsCompletelyPositive.

```
@[simp]
theorem id : (id A R).IsCompletelyPositive
```

theorem IsCompletelyPositive.add [source]

Sums of IsCompletelyPositive maps are IsCompletelyPositive.

```
theorem add {M₁ M₂ : MatrixMap A B R} (h₁ : M₁.IsCompletelyPositive) (h₂ : M₂.
    IsCompletelyPositive) :
    (M₁ + M₂).IsCompletelyPositive
```

theorem IsCompletelyPositive.smul [source]

Nonnegative scalings of `IsCompletelyPositive` maps are `IsCompletelyPositive`.

```
theorem smul {M : MatrixMap A B R} (hM : M.IsCompletelyPositive) {x : R} (hx : 0 ≤ x) :
    (x • M).IsCompletelyPositive
```

theorem zero [source]

The zero map `IsCompletelyPositive`.

```
theorem zero : (0 : MatrixMap A B R).IsCompletelyPositive
```

theorem finset_sum [source]

A finite sum of completely positive maps is completely positive.

```
theorem finset_sum {ι : Type*} [Fintype ι] {m : ι → MatrixMap A B R} (hm : ∀ i, (m i).
    IsCompletelyPositive) :
    (∑ i, m i).IsCompletelyPositive
```

theorem kron_kronecker_const [source]

The map that takes M and returns $M \otimes_k C$, where C is positive semidefinite, is a completely positive map.

```
theorem kron_kronecker_const {C : Matrix d d R} (h : C.PosSemidef) {h₁ h₂ : _} :
    IsCompletelyPositive
    (⟨⟨fun M => M ⊗ₖ C, h₁⟩, h₂⟩ : MatrixMap A (A × d) R)
```

theorem choi_of_kraus [source]

```
theorem choi_of_kraus (K : κ → Matrix B A 𝕜) :
    (MatrixMap.of_kraus K K).choi_matrix = ∑ k, Matrix.vecMulVec (fun (x : B × A) => K k x.1 x
    .2) (fun (x : B × A) => star (K k x.1 x.2))
```

def conj [source]

The linear map of conjugating a matrix by another, `x → y * x * y`^{H}.

```
@[simps]
def _root_.MatrixMap.conj (y : Matrix B A R) : MatrixMap A B R where
```

theorem conj_isPositive [source]

```
theorem conj_isPositive (M : Matrix B A 𝕜) : (conj M).IsPositive
```

theorem IsPositive_sum [source]

```
theorem IsPositive_sum {ι : Type*} [Fintype ι] (f : ι → MatrixMap A B ℂ) (h : ∀ i, (f i).
    IsPositive) :
    (∑ i, f i).IsPositive
```

theorem of_kraus_isPositive [source]

```
theorem of_kraus_isPositive (K : κ → Matrix B A ℂ) :
    (of_kraus K K).IsPositive
```

theorem conj_kron [source]

```
theorem conj_kron (M : Matrix B A 𝕜) (N : Matrix D C 𝕜) [DecidableEq C] :
    conj M ⊗ₖₘ conj N = conj (M ⊗ₖ N)
```

theorem congruence_one_eq_id [source]

```
theorem congruence_one_eq_id : conj (1 : Matrix A A ℂ) = MatrixMap.id A ℂ
```

theorem congruence_CP [source]

```
theorem congruence_CP {A B : Type*} [Fintype A] [Fintype B] [DecidableEq A] [DecidableEq B] (M
    : Matrix B A 𝕜) : (conj M).IsCompletelyPositive
```

theorem IsCompletelyPositive_sum [source]

```
theorem IsCompletelyPositive_sum {ι : Type*} [Fintype ι] (f : ι → MatrixMap A B ℂ) (h : ∀ i, (f
     i).IsCompletelyPositive) :
    (∑ i, f i).IsCompletelyPositive
```

theorem of_kraus_eq_sum_conj [source]

```
theorem of_kraus_eq_sum_conj (K : κ → Matrix B A 𝕜) :
    of_kraus K K = ∑ k, conj (K k)
```

theorem of_kraus_CP [source]

```
theorem of_kraus_CP (K : κ → Matrix B A 𝕜) : (of_kraus K K).IsCompletelyPositive
```

theorem exists_kraus_of_choi_PSD [source]

```
theorem exists_kraus_of_choi_PSD
    (C : Matrix (B × A) (B × A) 𝕜) (hC : C.PosSemidef) :
    ∃ (K : (B × A) → Matrix B A 𝕜), C = (MatrixMap.of_kraus K K).choi_matrix
```

theorem choi_matrix_eq_map_proj [source]

```
theorem choi_matrix_eq_map_proj (M : MatrixMap A B R) :
    M.choi_matrix = (M ⊗ₖₘ MatrixMap.id A R) (Matrix.vecMulVec (fun (x : A × A) => if x.1 = x.2
     then 1 else 0) (fun (x : A × A) => star (if x.1 = x.2 then 1 else 0)))
```

theorem choi_PSD_iff_CP_map [source]

Choi's theorem on completely positive maps: A map `IsCompletelyPositive` iff its Choi Matrix is PSD.

```
theorem choi_PSD_iff_CP_map (M : MatrixMap A B R) :
    M.IsCompletelyPositive ↔ M.choi_matrix.PosSemidef
```

theorem conj_eq_mulRightLinearMap_comp_mulRightLinearMap [source]

```
theorem conj_eq_mulRightLinearMap_comp_mulRightLinearMap (y : Matrix B A R) :
    conj y = mulRightLinearMap B R y.conjTranspose ∘ₗ mulLeftLinearMap A R y
```

theorem conj_isCompletelyPositive [source]

The act of conjugating (not necessarily by a unitary, just by any matrix at all) is completely positive.

```
theorem conj_isCompletelyPositive (M : Matrix B A R) : (conj M).IsCompletelyPositive
```

theorem IsCompletelyPositive.submatrix [source]

`MatrixMap.submatrix` is completely positive

```
theorem IsCompletelyPositive.submatrix (f : B → A) : (MatrixMap.submatrix R f).
    IsCompletelyPositive
```

theorem of_kraus_isCompletelyPositive [source]

The channel $X \mapsto \sum k : \kappa, (M\ k) * X * (M\ k)^H$ formed by Kraus operators $M : \kappa \to$ Matrix B A R is completely positive

```
theorem of_kraus_isCompletelyPositive (M : κ → Matrix B A R) :
    (of_kraus M M).IsCompletelyPositive
```

theorem choi_of_kraus_R [source]

The Choi matrix of a map in symmetric Kraus form is a sum of rank-1 projectors.

```
theorem choi_of_kraus_R [DecidableEq A] (K : κ → Matrix B A 𝕜) :
    (of_kraus K K).choi_matrix = ∑ k, Matrix.vecMulVec (fun (x : B × A) => K k x.1 x.2) (fun (x
     : B × A) => star (K k x.1 x.2))
```

theorem choi_eq_kron_id_apply_choi_id [source]

```
theorem choi_eq_kron_id_apply_choi_id (M : MatrixMap A B R) :
    M.choi_matrix = (M ⊗ₖₘ MatrixMap.id A R) ((MatrixMap.id A R).choi_matrix)
```

theorem choi_id_is_PSD [source]

```
theorem choi_id_is_PSD {A R : Type*} [Fintype A] [DecidableEq A] [RCLike R] :
    (MatrixMap.id A R).choi_matrix.PosSemidef
```

theorem is_CP_implies_choi_PSD [source]

```
theorem is_CP_implies_choi_PSD {A B R : Type*} [Fintype A] [Fintype B] [DecidableEq A] [
    DecidableEq B] [RCLike R] (M : MatrixMap A B R) (hCP : M.IsCompletelyPositive) :
    M.choi_matrix.PosSemidef
```

theorem exists_kraus [source]

```
theorem IsCompletelyPositive.exists_kraus (Φ : MatrixMap A B R) (hCP : Φ.IsCompletelyPositive)
    :
    ∃ (M : (B × A) → Matrix B A R), Φ = of_kraus M M
```

theorem kron_of_kraus [source]

The Kronecker product of two Kraus maps is the Kraus map of the Kronecker products of the operators.

```
theorem kron_of_kraus {A B C D R : Type*} [Fintype A] [Fintype B] [Fintype C] [Fintype D]
    [DecidableEq A] [DecidableEq C] [CommSemiring R] [StarRing R] [SMulCommClass R R R]
    {κ ι : Type*} [Fintype κ] [Fintype ι]
    (M : κ → Matrix B A R) (N : ι → Matrix D C R) :
    of_kraus M M ⊗ₖₘ of_kraus N N =
    of_kraus (fun (k : κ × ι) => M k.1 ⊗ₖ N k.2) (fun k => M k.1 ⊗ₖ N k.2)
```

theorem IsCompletelyPositive.kron [source]

The Kronecker product of IsCompletelyPositive maps is also completely positive.

```
theorem kron {D : Type*} [DecidableEq C] [Fintype D] {M₁ : MatrixMap A B R} {M₂ : MatrixMap C D
     R}
    (h₁ : M₁.IsCompletelyPositive) (h₂ : M₂.IsCompletelyPositive) : IsCompletelyPositive (M₁
    ⊗ₖₘ M₂)
```

theorem piProd [source]

The `MatrixMap.piProd` product of IsCompletelyPositive maps is also completely positive.

```
theorem piProd {Λi : ∀ i, MatrixMap (dI i) (dO i) R} (h₁ : ∀ i, (Λi i).IsCompletelyPositive) :
    IsCompletelyPositive (MatrixMap.piProd Λi)
```

2.35 QuantumInfo.Finite.Distance.Fidelity

QuantumInfo/Finite/Distance/Fidelity.lean on GitHub.

def fidelity [source]

The fidelity of two quantum states. This is the quantum version of the Bhattacharyya coefficient.

```
def fidelity (ρ σ : MState d) : ℝ
```

theorem fidelity_ge_zero [source]

```
theorem fidelity_ge_zero : 0 ≤ fidelity ρ σ
```

theorem fidelity_le_one [source]

```
theorem fidelity_le_one : fidelity ρ σ ≤ 1
```

def fidelity_prob [source]

The fidelity, as a `Prob` probability with value between 0 and 1.

```
def fidelity_prob : Prob
```

theorem fidelity_self_eq_one [source]

A state has perfect fidelity with itself.

```
theorem fidelity_self_eq_one : fidelity ρ ρ = 1
```

theorem fidelity_eq_one_iff_self [source]

The fidelity is 1 if and only if the two states are the same.

```
theorem fidelity_eq_one_iff_self : fidelity ρ σ = 1 ↔ ρ = σ
```

theorem fidelity_symm [source]

The fidelity is a symmetric quantity.

```
theorem fidelity_symm : fidelity ρ σ = fidelity σ ρ
```

theorem fidelity_channel_nondecreasing [source]

The fidelity cannot decrease under the application of a channel.

```
theorem fidelity_channel_nondecreasing [DecidableEq d₂] (Λ : CPTPMap d d₂) : fidelity (Λ ρ) (Λ
    σ) ≥ fidelity ρ σ
```

2.36 QuantumInfo.Finite.Distance.TraceDistance

QuantumInfo/Finite/Distance/TraceDistance.lean on GitHub.

def TrDistance [source]

The trace distance between two quantum states: half the trace norm of the difference (ρ - σ).

```
def TrDistance (ρ σ : MState d) : ℝ
```

theorem ge_zero [source]

```
theorem ge_zero : 0 ≤ TrDistance ρ σ
```

theorem le_one [source]

```
theorem le_one : TrDistance ρ σ ≤ 1
```

def prob [source]

The trace distance, as a `Prob` probability with value between 0 and 1.

```
def prob : Prob
```

theorem symm [source]

The trace distance is a symmetric quantity.

```
theorem symm : TrDistance ρ σ = TrDistance σ ρ
```

theorem eq_abs_eigenvalues [source]

The trace distance is equal to half the 1-norm of the eigenvalues of their difference .

```
theorem eq_abs_eigenvalues : TrDistance ρ σ = (1/2:ℝ) *
    ∑ i, abs ((ρ.Hermitian.sub σ.Hermitian).eigenvalues i)
```

2.37 QuantumInfo.Finite.Ensemble

QuantumInfo/Finite/Ensemble.lean on GitHub.

abbrev MEnsemble [source]

A mixed-state ensemble is a random variable valued in `MState d`. That is, a collection of mixed states `var : α → MState d`, each with their own probability weight described by `distr : ProbDistribution α`.

```
abbrev MEnsemble (d : Type*) (α : Type*) [Fintype d] [DecidableEq d] [Fintype α]
```

abbrev PEnsemble [source]

A pure-state ensemble is a random variable valued in `Ket d`. That is, a collection of pure states `var : α → Ket d`, each with their own probability weight described by `distr : ProbDistribution α`.

```
abbrev PEnsemble (d : Type*) (α : Type*) [Fintype d] [Fintype α]
```

abbrev MEnsemble.states [source]

Alias for `ProbDistribution.var` for mixed-state ensembles.

```
abbrev MEnsemble.states [Fintype α] : MEnsemble d α → (α → MState d)
```

abbrev PEnsemble.states [source]

Alias for `ProbDistribution.var` for pure-state ensembles.

```
abbrev PEnsemble.states [Fintype α] : PEnsemble d α → (α → Ket d)
```

instance Coe (PEnsemble d α) (MEnsemble d α) [source]

A pure-state ensemble is a mixed-state ensemble if all kets are interpreted as mixed states.

```
instance : Coe (PEnsemble d α) (MEnsemble d α)
```

theorem toMEnsemble_mk [source]

```
@[simp]
theorem toMEnsemble_mk : (toMEnsemble ⟨ps, distr⟩ : MEnsemble d α) = ⟨pure ∘ ps, distr⟩
```

theorem coe_PEnsemble_iff_pure_states [source]

A mixed-state ensemble comes from a pure-state ensemble if and only if all states are pure.

```
theorem coe_PEnsemble_iff_pure_states (me : MEnsemble d α): (∃ pe : PEnsemble d α, ↑pe = me) ↔
    (∃ ψ : α → Ket d, me.states = MState.pure ∘ ψ)
```

def mix [source]

The resulting mixed state after mixing the states in an ensemble with their respective probability weights. Note that, generically, a single mixed state has infinitely many ensembles that mixes into it.

```
def mix (e : MEnsemble d α) : MState d
```

theorem mix_of [source]

```
@[simp]
theorem mix_of (e : MEnsemble d α) : (mix e).m = ∑ i, (e.distr i : ℝ) • (e.states i).m
```

def congrMEnsemble [source]

Two mixed-state ensembles indexed by `\{}alpha` and `\{}beta` are equivalent if `α ≃ β`.

```
def congrMEnsemble (σ : α ≃ β) : MEnsemble d α ≃ MEnsemble d β
```

def congrPEnsemble [source]

Two pure-state ensembles indexed by `\{}alpha` and `\{}beta` are equivalent if `α ≃ β`.

```
def congrPEnsemble (σ : α ≃ β) : PEnsemble d α ≃ PEnsemble d β
```

theorem mix_congrMEnsemble_eq_mix [source]

Equivalence of mixed-state ensembles leaves the resulting mixed state invariant

```
@[simp]
theorem mix_congrMEnsemble_eq_mix (σ : α ≃ β) (e : MEnsemble d α) : mix (congrMEnsemble σ e) =
    mix e
```

theorem mix_congrPEnsemble_eq_mix [source]

Equivalence of pure-state ensembles leaves the resulting mixed state invariant

```
@[simp]
theorem mix_congrPEnsemble_eq_mix (σ : α ≃ β) (e : PEnsemble d α) : mix (toMEnsemble (
    congrPEnsemble σ e)) = mix (↑e : MEnsemble d α)
```

def average [source]

The average of a function `f : MState d → T`, where `T` is of `Mixable U T` instance, on a mixed-state ensemble `e` is the expectation value of `f` acting on the states of `e`, with the corresponding probability weights from `e.distr`.

```
def average {T : Type _} {U : Type*} [AddCommGroup U] [Module ℝ U] [inst : Mixable U T] (f :
    MState d → T) (e : MEnsemble d α) : T
```

def average_NNReal [source]

A version of `average` conveniently specialized for functions `f : MState d → ℝ≥0` returning nonnegative reals. Notably, the average is also a nonnegative real number.

```
def average_NNReal {d : Type _} [Fintype d] [DecidableEq d] (f : MState d → NNReal) (e :
    MEnsemble d α) : NNReal
```

def pure_average [source]

The average of a function `f : Ket d → T`, where `T` is of `Mixable U T` instance, on a pure-state ensemble `e` is the expectation value of `f` acting on the states of `e`, with the corresponding probability weights from `e.distr`.

```
def pure_average {T : Type _} {U : Type*} [AddCommGroup U] [Module ℝ U] [inst : Mixable U T] (f
    : Ket d → T) (e : PEnsemble d α) : T
```

def pure_average_NNReal [source]

A version of `average` conveniently specialized for functions `f : Ket d → ℝ≥0` returning nonnegative reals. Notably, the average is also a nonnegative real number.

```
def pure_average_NNReal {d : Type _} [Fintype d] (f : Ket d → NNReal) (e : PEnsemble d α) :
    NNReal
```

theorem average_of_pure_ensemble [source]

The average of `f : MState d → T` on a coerced pure-state ensemble `↑e : MEnsemble d α` is equal to averaging the restricted function over Kets `f ∘ pure : Ket d → T` on `e`.

```
theorem average_of_pure_ensemble {T : Type _} {U : Type*} [AddCommGroup U] [Module ℝ U] [inst :
    Mixable U T]
  (f : MState d → T) (e : PEnsemble d α) :
  average f (toMEnsemble e) = pure_average (f ∘ pure) e
```

theorem distr_toMEnsemble [source]

```
@[simp]
theorem distr_toMEnsemble (e : PEnsemble d α) : (toMEnsemble e).distr = e.distr
```

theorem mix_pEnsemble_pure_iff_pure [source]

```
theorem mix_pEnsemble_pure_iff_pure {e : PEnsemble d α} :
    mix (toMEnsemble e) = MState.pure ψ ↔
    ∀ i : α, e.distr i ≠ 0 → MState.pure (e.states i) = MState.pure ψ
```

theorem mix_pEnsemble_pure_average [source]

The average of `f : Ket d → T` on an ensemble that mixes to a pure state `ψ` is `f ψ`

```
theorem mix_pEnsemble_pure_average {e : PEnsemble d α} {T : Type _} {U : Type*} [AddCommGroup U
    ] [Module ℝ U]
    [inst : Mixable U T] (f : Ket d → T) (hf : ∀ ψ φ, Ket.PhaseEquiv.r ψ φ → f ψ = f φ)
    (hmix : mix (toMEnsemble e) = MState.pure ψ) :
  pure_average f e = f ψ
```

theorem sum_prob_mul_eq_one_iff [source]

```
theorem sum_prob_mul_eq_one_iff {ι : Type*} [Fintype ι] (p : ι → ℝ) (x : ι → ℝ)
    (hp : ∀ i, 0 ≤ p i) (hsum : ∑ i, p i = 1) (hx : ∀ i, x i ≤ 1) :
    (∑ i, p i * x i = 1) ↔ ∀ i, p i ≠ 0 → x i = 1
```

theorem exp_val_pure_eq_one_iff [source]

```
theorem MState.exp_val_pure_eq_one_iff {d : Type*} [Fintype d] [DecidableEq d] (ρ : MState d) (
    ψ : Ket d) :
    ρ.exp_val (pure ψ) = 1 ↔ ρ = pure ψ
```

theorem mix_mEnsemble_pure_iff_pure [source]

```
theorem mix_mEnsemble_pure_iff_pure {e : MEnsemble d α} :
    mix e = pure ψ ↔ ∀ i : α, e.distr i ≠ 0 → e.states i = MState.pure ψ
```

theorem mix_mEnsemble_pure_average [source]

The average of `f : MState d → T` on an ensemble that mixes to a pure state `ψ` is `f (pure ψ)`

```
theorem mix_mEnsemble_pure_average {e : MEnsemble d α} {T : Type _} {U : Type*} [AddCommGroup U
    ] [Module ℝ U] [inst : Mixable U T] (f : MState d → T) (hmix : mix e = pure ψ) :
  average f e = f (pure ψ)
```

def trivial_mEnsemble [source]

The trivial mixed-state ensemble of `ρ` consists of copies of `rho`, with the `i`-th one having probability 1.

```
def trivial_mEnsemble (ρ : MState d) (i : α) : MEnsemble d α
```

theorem trivial_mEnsemble_mix [source]

The trivial mixed-state ensemble of `ρ` mixes to `ρ`

```
theorem trivial_mEnsemble_mix (ρ : MState d) : ∀ i : α, mix (trivial_mEnsemble ρ i) = ρ
```

theorem trivial_mEnsemble_average [source]

The average of `f : MState d → T` on a trivial ensemble of `ρ` is `f ρ`

```
theorem trivial_mEnsemble_average {T : Type _} {U : Type*} [AddCommGroup U] [Module ℝ U] [inst
    : Mixable U T] (f : MState d → T) (ρ : MState d):
  ∀ i : α, average f (trivial_mEnsemble ρ i) = f ρ
```

instance MEnsemble.instInhabited [source]

```
instance MEnsemble.instInhabited [Nonempty d] [Inhabited α] : Inhabited (MEnsemble d α) where
```

def trivial_pEnsemble [source]

The trivial pure-state ensemble of `ψ` consists of copies of `ψ`, with the `i`-th one having probability 1.

```
def trivial_pEnsemble (ψ : Ket d) (i : α) : PEnsemble d α
```

theorem trivial_pEnsemble_mix [source]

The trivial pure-state ensemble of ψ mixes to ψ

```
theorem trivial_pEnsemble_mix : ∀ i : α, mix (toMEnsemble (trivial_pEnsemble ψ i)) = MState.
    pure ψ
```

theorem trivial_pEnsemble_average [source]

The average of `f : Ket d → T` on a trivial ensemble of ψ is `f ψ`

```
theorem trivial_pEnsemble_average {T : Type _} {U : Type*} [AddCommGroup U] [Module ℝ U] [inst
    : Mixable U T] (f : Ket d → T) :
  ∀ i : α, pure_average f (trivial_pEnsemble ψ i) = f ψ
```

instance PEnsemble.instInhabited [source]

```
instance PEnsemble.instInhabited [Nonempty d] [Inhabited α] : Inhabited (PEnsemble d α) where
```

def spectral_ensemble [source]

The spectral pure-state ensemble of ρ. The states are its eigenvectors, and the probabilities, eigenvalues.

```
def spectral_ensemble (ρ : MState d) : PEnsemble d d where
```

theorem spectral_decomposition_sum [source]

```
theorem spectral_decomposition_sum {d 𝕜 : Type*} [Fintype d] [DecidableEq d] [RCLike 𝕜]
    {A : Matrix d d 𝕜} (hA : A.IsHermitian) :
    A = ∑ i, (hA.eigenvalues i) • (Matrix.vecMulVec (hA.eigenvectorBasis i) (star (hA.
    eigenvectorBasis i)))
```

theorem spectral_ensemble_mix [source]

The spectral pure-state ensemble of ρ mixes to ρ

```
theorem spectral_ensemble_mix {ρ : MState d} : mix (↑(spectral_ensemble ρ) : MEnsemble d d) = ρ
```

2.38 QuantumInfo.Finite.Entanglement

QuantumInfo/Finite/Entanglement.lean on GitHub.

def convex_roof_ENNReal [source]

Convex roof extension of a function `g : KetUpToPhase d → ℝ≥0`, defined as the infimum of all pure-state ensembles of a given ρ of the average of `g` in that ensemble.

This is valued in the extended nonnegative real numbers `ℝ≥0∞` to have good properties of the infimum, which come from the fact that `ℝ≥0∞` is a complete lattice. For example, it is necessary for `le_iInf` and `iInf_le_of_le`. However, it is also proven in `convex_roof_ne_top` that the convex roof is never ∞, so the definition `convex_roof` should be used in most applications.

```
def convex_roof_ENNReal {d : Type _} [Fintype d] [DecidableEq d] (g : KetUpToPhase d → ℝ≥0) :
    MState d → ℝ≥0∞
```

def mixed_convex_roof_ENNReal [source]

Mixed convex roof extension of a function `f : MState d → ℝ≥0`, defined as the infimum of all mixed-state ensembles of a given `ρ` of the average of `f` on that ensemble.

This is valued in the extended nonnegative real numbers `ℝ≥0∞` to have good properties of the infimum, which come from the fact that `ℝ≥0∞` is a complete lattice (see `ENNReal.instCompleteLinearOrder`). However, it is also proven in `mixed_convex_roof_ne_top` that the mixed convex roof is never ∞, so the definition `mixed_convex_roof` should be used in most applications.

```
def mixed_convex_roof_ENNReal {d : Type _} [Fintype d] [DecidableEq d] (f : MState d → ℝ≥0) :
    MState d → ℝ≥0∞
```

theorem convex_roof_ne_top [source]

The convex roof extension `convex_roof_ENNReal` is never ∞.

```
theorem convex_roof_ne_top : ∀ ρ, convex_roof_ENNReal g ρ ≠ ∞
```

theorem mixed_convex_roof_ne_top [source]

The convex roof extension `mixed_convex_roof_ENNReal` is never ∞.

```
theorem mixed_convex_roof_ne_top : ∀ ρ, mixed_convex_roof_ENNReal f ρ ≠ ∞
```

def convex_roof [source]

Convex roof extension of a function `g : KetUpToPhase d → ℝ≥0`, defined as the infimum of all pure-state ensembles of a given `ρ` of the average of `g` in that ensemble.

This is valued in the nonnegative real numbers `ℝ≥0` by applying `ENNReal.toNNReal` to `convex_roof_ENNReal`. Hence, it should be used in proofs alongside `convex_roof_ne_top`.

```
def convex_roof : MState d → ℝ≥0
```

def mixed_convex_roof [source]

Mixed convex roof extension of a function `f : MState d → ℝ≥0`, defined as the infimum of all mixed-state ensembles of a given `ρ` of the average of `f` on that ensemble.

This is valued in the nonnegative real numbers `ℝ≥0` by applying `ENNReal.toNNReal` to `mixed_convex_roof_ENNReal`. Hence, it should be used in proofs alongside `mixed_convex_roof_ne_top`.

```
def mixed_convex_roof : MState d → ℝ≥0
```

def convex_roof_of_MState_fun [source]

Auxiliary function. Convex roof of a function `f : MState d → ℝ≥0` defined over mixed states by restricting `f` to pure states, via `pureQ : KetUpToPhase d → MState d`.

```
def convex_roof_of_MState_fun : MState d → ℝ≥0
```

theorem le_mixed_convex_roof [source]

```
theorem le_mixed_convex_roof (ρ : MState d) :
  (∀ n > 0, ∀ e : MEnsemble d (Fin n), mix e = ρ → c ≤ average_NNReal f e) → (c ≤
    mixed_convex_roof f ρ)
```

theorem le_convex_roof [source]

```
theorem le_convex_roof (ρ : MState d) :
  (∀ n > 0, ∀ e : PEnsemble d (Fin n), mix (toMEnsemble e) = ρ → c ≤ pure_average_NNReal (g ∘
    KetUpToPhase.mk) e) → (c ≤ convex_roof g ρ)
```

theorem convex_roof_le [source]

```
theorem convex_roof_le (ρ : MState d):
(∃ n > 0, ∃ e : PEnsemble d (Fin n), mix (toMEnsemble e) = ρ ∧ pure_average_NNReal (g ∘
    KetUpToPhase.mk) e ≤ c) → (convex_roof g ρ ≤ c)
```

theorem mixed_convex_roof_le [source]

```
theorem mixed_convex_roof_le (ρ : MState d):
(∃ n > 0, ∃ e : MEnsemble d (Fin n), mix e = ρ ∧ average_NNReal f e ≤ c) → (mixed_convex_roof f
     ρ ≤ c)
```

theorem mixed_convex_roof_le_convex_roof [source]

The mixed convex roof extension of `f` is smaller than or equal to its convex roof extension, since the former minimizes over a larger set of ensembles.

```
theorem mixed_convex_roof_le_convex_roof : mixed_convex_roof f ≤ convex_roof_of_MState_fun f
```

theorem convex_roof_of_pure [source]

The convex roof extension of `g : KetUpToPhase d → ℝ≥0` applied to a pure state `ψ` is `g (KetUpToPhase.mk ψ)`.

```
theorem convex_roof_of_pure (ψ : Ket d) : convex_roof g (pure ψ) = g (KetUpToPhase.mk ψ)
```

theorem mixed_convex_roof_of_pure [source]

The mixed convex roof extension of `f : MState d → ℝ≥0` applied to a pure state `ψ` is `f (pure ψ)`.

```
theorem mixed_convex_roof_of_pure (ψ : Ket d) : mixed_convex_roof f (pure ψ) = f (pure ψ)
```

def EoF [source]

Entanglement of Formation of bipartite systems. It is the convex roof extension of the von Neumann entropy of one of the subsystems (here chosen to be the left one, but see `Entropy.S_von_of_partial_eq`).

The function $S_{vn} \circ$ `traceRight` $\circ$ `pure` is phase-invariant because `pure` maps phase-equivalent kets to the same mixed state, so it descends to `KetUpToPhase` via `pureQ`.

```
def EoF : MState (d₁ × d₂) → ℝ≥0
```

theorem traceRight_pure_MES [source]

```
theorem traceRight_pure_MES (d : Type*) [Fintype d] [DecidableEq d] [Nonempty d] :
    (MState.pure (Ket.MES d)).traceRight = MState.uniform
```

theorem S_{vn}_eq_trace_cfc [source]

```
theorem S_vn_eq_trace_cfc {d : Type*} [Fintype d] [DecidableEq d] (ρ : MState d) :
    S_vn ρ = (HermitianMat.cfc ρ.M Real.negMulLog).trace
```

theorem S_{vn}_ofClassical [source]

```
theorem S_vn_ofClassical {d : Type*} [Fintype d] [DecidableEq d] (dist : ProbDistribution d) :
    S_vn (MState.ofClassical dist) = H_s dist
```

theorem EoF_of_MES [source]

The entanglement of formation of the maximally entangled state with on-site dimension is log().

```
theorem EoF_of_MES : EoF (pure <| Ket.MES d) = Real.log (Finset.card Finset.univ (α := d))
```

2.39 QuantumInfo.Finite.Entropy.DPI

QuantumInfo/Finite/Entropy/DPI.lean on GitHub.

def weighted_norm [source]

The weighted norm \{}|X\{}|_{p, σ} defined in the paper.

```
noncomputable def weighted_norm (p : ℝ) (σ : MState d) (X : Matrix d d ℂ) : ℝ
```

def spectralNorm_mat [source]

The spectral norm (operator norm) of a matrix.

```
noncomputable def spectralNorm_mat (A : Matrix d d ℂ) : ℝ
```

def weighted_norm_infty [source]

The weighted norm for $p =$ \{}infty.

```
noncomputable def weighted_norm_infty (_ : MState d) (X : Matrix d d ℂ) : ℝ
```

def Gamma [source]

The map $\Gamma_\sigma(X) = \sigma^{1/2} X \sigma^{1/2}$.

```
noncomputable def Gamma (σ : MState d) (X : Matrix d d ℂ) : Matrix d d ℂ
```

def Gamma_inv [source]

The inverse map $\Gamma_\sigma^{-1}(X) = \sigma^{-1/2} X \sigma^{-1/2}$.

```
noncomputable def Gamma_inv (σ : MState d) (X : Matrix d d ℂ) : Matrix d d ℂ
```

def T_op [source]

The operator $T = \Gamma_{\Phi(\sigma)}^{-1} \circ \Phi \circ \Gamma_\sigma$.

```
noncomputable def T_op (Φ : CPTPMap d d₂) (σ : MState d) (X : Matrix d d ℂ) : Matrix d₂ d₂ ℂ
```

def induced_norm [source]

The induced norm of a map Ψ: M_d -> M_d2 with respect to weighted norms.

```
noncomputable def induced_norm (p : ℝ) (σ : MState d) (Φ : CPTPMap d d₂) (Ψ : Matrix d d ℂ →
    Matrix d₂ d₂ ℂ) : ℝ
```

def induced_norm_infty_map [source]

```
noncomputable def induced_norm_infty_map (σ : MState d) (Φ : CPTPMap d d₂) (Ψ : Matrix d d ℂ →
    Matrix d₂ d₂ ℂ) : ℝ
```

def T_map [source]

```
noncomputable def T_map (σ : MState d) (Φ : CPTPMap d d₂) : MatrixMap d d₂ ℂ
```

def Gamma_map [source]

```
noncomputable def Gamma_map (σ : MState d) : MatrixMap d d ℂ
```

lemma Gamma_map_eq [source]

```
lemma Gamma_map_eq (σ : MState d) (X : Matrix d d ℂ) :
    Gamma_map σ X = Gamma σ X
```

lemma Gamma_map_CP [source]

```
lemma Gamma_map_CP (σ : MState d) : (Gamma_map σ).IsCompletelyPositive
```

def Gamma_inv_map [source]

```
noncomputable def Gamma_inv_map (σ : MState d) : MatrixMap d d ℂ
```

lemma Gamma_inv_map_eq [source]

```
lemma Gamma_inv_map_eq (σ : MState d) (X : Matrix d d ℂ) :
    Gamma_inv_map σ X = Gamma_inv σ X
```

def sigma_inv_sqrt [source]

```
noncomputable def sigma_inv_sqrt (σ : MState d) : Matrix d d ℂ
```

lemma Gamma_inv_map_eq_conj [source]

```
lemma Gamma_inv_map_eq_conj (σ : MState d) :
    Gamma_inv_map σ = MatrixMap.conj (sigma_inv_sqrt σ)
```

lemma Gamma_inv_map_CP [source]

```
lemma Gamma_inv_map_CP (σ : MState d) : (Gamma_inv_map σ).IsCompletelyPositive
```

lemma T_map_eq_comp [source]

```
lemma T_map_eq_comp (σ : MState d) (Φ : CPTPMap d d₂) :
    T_map σ Φ = (Gamma_inv_map (Φ σ)).comp (Φ.map.comp (Gamma_map σ))
```

lemma T_is_CP [source]

```
lemma T_is_CP (σ : MState d) (Φ : CPTPMap d d₂) :
    (T_map σ Φ).IsCompletelyPositive
```

lemma T_is_positive [source]

```
lemma T_is_positive (σ : MState d) (Φ : CPTPMap d d₂) :
    (T_map σ Φ).IsPositive
```

lemma weighted_norm_one_eq_trace_norm_Gamma [source]

```
lemma weighted_norm_one_eq_trace_norm_Gamma (σ : MState d) (X : Matrix d d ℂ) :
    weighted_norm 1 σ X = schattenNorm (Gamma σ X) 1
```

def general_induced_norm [source]

```
noncomputable def general_induced_norm
    (p q : ℝ) (σ : MState d) (σ' : MState d₂)
    (Ψ : MatrixMap d d₂ ℂ) : ℝ
```

lemma cfc_mul [source]

```
lemma HermitianMat.cfc_mul {d : Type*} [Fintype d] [DecidableEq d]
    (A : HermitianMat d ℂ) (f g : ℝ → ℝ) :
    (A.cfc f).mat * (A.cfc g).mat = (A.cfc (fun x => f x * g x)).mat
```

lemma Gamma_one [source]

```
lemma Gamma_one (σ : MState d) : Gamma σ 1 = σ.M.mat
```

lemma Gamma_inv_self [source]

```
lemma Gamma_inv_self (σ : MState d) (hσ : σ.m.PosDef) :
    Gamma_inv σ σ.M.mat = 1
```

lemma CPTPMap_apply_MState_M [source]

```
lemma CPTPMap_apply_MState_M (Φ : CPTPMap d d₂) (σ : MState d) :
    (Φ σ).M.mat = Φ.map σ.M.mat
```

theorem T_map_unital [source]

```
theorem T_map_unital (σ : MState d) (Φ : CPTPMap d d₂) (hΦσ : (Φ σ).m.PosDef) :
    (T_map σ Φ) 1 = 1
```

theorem T_map_is_CP_proof [source]

```
theorem T_map_is_CP_proof (σ : MState d) (Φ : CPTPMap d d₂) :
    (T_map σ Φ).IsCompletelyPositive
```

lemma Gamma_Gamma_inv [source]

```
lemma Gamma_Gamma_inv (σ : MState d) (hσ : σ.m.PosDef) (X : Matrix d d ℂ) :
    Gamma σ (Gamma_inv σ X) = X
```

theorem le_smul_one_imp_eigenvalues_le [source]

```
theorem HermitianMat.le_smul_one_imp_eigenvalues_le (A : HermitianMat d ℂ) (M : ℝ)
    (h : A ≤ M • (1 : HermitianMat d ℂ)) (i : d) :
    A.H.eigenvalues i ≤ M
```

theorem eigenvalues_le_imp_le_smul_one [source]

```
theorem HermitianMat.eigenvalues_le_imp_le_smul_one (A : HermitianMat d ℂ) (M : ℝ)
    (h : ∀ i, A.H.eigenvalues i ≤ M) :
    A ≤ M • (1 : HermitianMat d ℂ)
```

theorem block_matrix_posSemidef [source]

The block matrix [[1, X], [X†, X†X]] is positive semidefinite.

```
theorem block_matrix_posSemidef {m n k : Type*} [Fintype m] [Fintype n] [Fintype k]
    (X : Matrix k n ℂ) (Y : Matrix k m ℂ):
    (Matrix.fromBlocks (Y^H * Y) (Y^H * X) (X^H * Y) (X^H * X)).PosSemidef
```

theorem block_matrix_one_posSemidef [source]

```
theorem block_matrix_one_posSemidef {m n : Type*} [Fintype m] [Fintype n] [DecidableEq m]
    (X : Matrix m n ℂ) :
    (Matrix.fromBlocks 1 X X^H (X^H * X)).PosSemidef
```

theorem sandwichedRenyiEntropy_DPI [source]

The Data Processing Inequality for the Sandwiched Renyi relative entropy. Proved in `https://arxiv.org/pdf/1306.5920`. Seems kind of involved.

```
theorem sandwichedRenyiEntropy_DPI (hα : 1 ≤ α) (ρ σ : MState d) (Φ : CPTPMap d d₂) :
    D̃ α(Φ ρ‖Φ σ) ≤ D̃ α(ρ‖σ)
```

axiom sandwichedRenyiEntropy_DPI_ax [source]

```
axiom sandwichedRenyiEntropy_DPI_ax : type_of% @sandwichedRenyiEntropy_DPI
```

2.40 QuantumInfo.Finite.Entropy.Relative

`QuantumInfo/Finite/Entropy/Relative.lean` on GitHub.

theorem sandwiched_trace_pos [source]

```
private theorem sandwiched_trace_pos (h : σ.M.ker ≤ ρ.M.ker) :
    0 < ((ρ.M.conj (σ.M ^ ((1 - α)/(2 * α)) ).mat) ^ α).trace
```

lemma weighted_jensen_rpow [source]

*Weighted Jensen inequality: for weights $w_j \geq 0$ with $\sum w_j = 1$, values $b_j \geq 0$, and $q \geq 1$: (∑_j w_j * b_j)^q ≤ ∑_j w_j * b_j^q.*

This is the special case of `Real.rpow_arith_mean_le_arith_mean_rpow` applied to `Finset.univ`

```
lemma weighted_jensen_rpow (b w : d → ℝ) (q : ℝ)
  (hb : ∀ j, 0 ≤ b j) (hw : ∀ j, 0 ≤ w j) (hsum : ∑ j, w j = 1) (hq : 1 ≤ q) :
    (∑ j, w j * b j) ^ q ≤ ∑ j, w j * b j ^ q
```

lemma doubly_stochastic_holder [source]

*Doubly stochastic Hölder inequality: for nonneg a, b, doubly stochastic w, and conjugate p, q > 1: ∑_{ij} a_i * b_j * w_{ij} ≤ (∑ a_i^p)^{1/p} * (∑ b_j^q)^{1/q}.*

```
lemma doubly_stochastic_holder (a b : d → ℝ) (w : d → d → ℝ)
    (ha : ∀ i, 0 ≤ a i) (hb : ∀ j, 0 ≤ b j)
    (hw : ∀ i j, 0 ≤ w i j)
    (hrow : ∀ i, ∑ j, w i j = 1) (hcol : ∀ j, ∑ i, w i j = 1)
    (p q : ℝ) (hp : 1 < p) (hpq : 1/p + 1/q = 1) :
    ∑ i, ∑ j, a i * b j * w i j ≤ (∑ i, a i ^ p) ^ (1/p) * (∑ j, b j ^ q) ^ (1/q)
```

lemma inner_le_trace_rpow_mul [source]

*Hermitian trace Hölder inequality: for PSD A, B and conjugate exponents p, q > 1, ⟪A, B⟫ ≤ Tr[A^p]^(1/p) * Tr[B^q]^(1/q).*

```
lemma HermitianMat.inner_le_trace_rpow_mul
    {A B : HermitianMat d ℂ} (hA : 0 ≤ A) (hB : 0 ≤ B)
    (p q : ℝ) (hp : 1 < p) (hpq : 1/p + 1/q = 1) :
    ⟪A, B⟫_ℝ ≤ (A ^ p).trace ^ (1/p) * (B ^ q).trace ^ (1/q)
```

lemma rpow_le_one' [source]

```
lemma MState.rpow_le_one' {r : ℝ} (hσ : 0 < r) : σ.M ^ r ≤ 1
```

lemma eigenvalues_le_one_of_le_one [source]

If $A \geq 0$ and $A \leq 1$, then each eigenvalue of A is in $[0, 1]$.

```
lemma HermitianMat.eigenvalues_le_one_of_le_one
    (A : HermitianMat d ℂ) (hA1 : A ≤ 1) (i : d) :
    A.H.eigenvalues i ≤ 1
```

lemma trace_rpow_le_trace_of_le_one [source]

For positive $A \leq 1$ and $p \geq 1$, `Tr[A^p] ≤ Tr[A]`.

```
lemma HermitianMat.trace_rpow_le_trace_of_le_one
    (A : HermitianMat d ℂ) (hA : 0 ≤ A) (hA1 : A ≤ 1)
    (p : ℝ) (hp : 1 ≤ p) :
    (A ^ p).trace ≤ A.trace
```

lemma trace_conj_rpow_eq_inner [source]

```
private lemma trace_conj_rpow_eq_inner (hα₀ : 0 < α) (hα : α < 1) :
    ((ρ.M ^ α).conj (σ.M ^ ((1 - α) / (2 * α) * α)).mat).trace = ⟪ρ.M ^ α, σ.M ^ (1 - α)⟫_ℝ
```

lemma inner_rpow_le_one [source]

```
private lemma inner_rpow_le_one (hα₀ : 0 < α) (hα : α < 1) :
    ⟪ρ.M ^ α, σ.M ^ (1 - α)⟫_ℝ ≤ 1
```

theorem sandwiched_trace_of_lt_1 [source]

```
private theorem sandwiched_trace_of_lt_1 (hα₀ : 0 < α) (hα : α < 1) :
    ((ρ.M.conj (σ.M ^ ((1 - α)/(2 * α)) ).mat) ^ α).trace ≤ 1
```

lemma rpow_neg_mul_rpow_eq_supportProj [source]

For PSD A and $p \neq 0$, `A^{-p} * A^p = HermitianMat.supportProj A`.

```
lemma HermitianMat.rpow_neg_mul_rpow_eq_supportProj
    {A : HermitianMat d ℂ} (hA : 0 ≤ A) {p : ℝ} (hp : p ≠ 0) :
    (A ^ (-p)).mat * (A ^ p).mat = A.supportProj.mat
```

lemma supportProj_mul_self [source]

```
lemma HermitianMat.supportProj_mul_self (A : HermitianMat d ℂ) :
    A.supportProj.mat * A.mat = A.mat
```

lemma inner_supportProj_self [source]

```
lemma HermitianMat.inner_supportProj_self (A : HermitianMat d ℂ) :
    ⟪A, A.supportProj⟫ = A.trace
```

lemma mul_supportProj_of_ker_le [source]

```
lemma HermitianMat.mul_supportProj_of_ker_le {A B : HermitianMat d ℂ}
  (h : LinearMap.ker B.lin.toLinearMap ≤ LinearMap.ker A.lin.toLinearMap) :
    A.mat * B.supportProj.mat = A.mat
```

lemma inner_supportProj_of_ker_le [source]

```
lemma HermitianMat.inner_supportProj_of_ker_le {A B : HermitianMat d ℂ}
  (h : LinearMap.ker B.lin.toLinearMap ≤ LinearMap.ker A.lin.toLinearMap) :
    ⟪A, B.supportProj⟫ = A.trace
```

lemma supportProj_inner_density [source]

```
lemma supportProj_inner_density (h : σ.M.ker ≤ ρ.M.ker) :
    ⟪σ.M.supportProj, ρ.M⟫_ℝ = 1
```

lemma sandwiched_inner_eq_one [source]

```
private lemma sandwiched_inner_eq_one (h : σ.M.ker ≤ ρ.M.ker) (t : ℝ) :
    ⟪ρ.M.conj (σ.M ^ t).mat, σ.M ^ (-2 * t)⟫_ℝ = 1
```

theorem sandwiched_trace_of_gt_1 [source]

```
private theorem sandwiched_trace_of_gt_1 (h : σ.M.ker ≤ ρ.M.ker) (hα : α > 1) :
    1 ≤ ((ρ.M.conj (σ.M ^ ((1 - α)/(2 * α)) ).mat) ^ α).trace
```

theorem sandwichedRelRentropy_nonneg_α_lt_1 [source]

```
private theorem sandwichedRelRentropy_nonneg_α_lt_1 (h : σ.M.ker ≤ ρ.M.ker) (hα0 : 0 < α) (hα :
     α < 1) :
    0 ≤ ((ρ.M.conj (σ.M ^ ((1 - α)/(2 * α)) ).mat) ^ α).trace.log / (α - 1)
```

theorem sandwichedRelRentropy_nonneg_α_gt_1 [source]

```
private theorem sandwichedRelRentropy_nonneg_α_gt_1 (h : σ.M.ker ≤ ρ.M.ker) (hα : α > 1) :
    0 ≤ ((ρ.M.conj (σ.M ^ ((1 - α)/(2 * α)) ).mat) ^ α).trace.log / (α - 1)
```

lemma trace_at_one [source]

```
private lemma sandwichedRelRentropy.trace_at_one (ρ σ : MState d) :
    ((ρ.M.conj (σ.M ^ ((1 - (1:ℝ)) / (2 * (1:ℝ))))).mat) ^ (1:ℝ)).trace = 1
```

lemma hasDerivAt_trace_rpow_at_one [source]

```
private lemma hasDerivAt_trace_rpow_at_one (B : HermitianMat d ℂ) (hB : 0 ≤ B) :
    HasDerivAt (fun α : ℝ => (B ^ α).trace) ⟪B, B.log⟫ 1
```

lemma trace_conj_eq_inner_rpow [source]

```
private lemma trace_conj_eq_inner_rpow {ρ σ : MState d} {t : ℝ} (ht : t ≠ 0) :
    (ρ.M.conj (σ.M ^ t).mat).trace = ⟪ρ.M, σ.M ^ (2 * t)⟫
```

def eigenWeight [source]

```
private def eigenWeight (ρ σ : MState d) (i : d) : ℝ
```

lemma inner_cfc_eq_sum_eigenWeight [source]

```
private lemma inner_cfc_eq_sum_eigenWeight (ρ σ : MState d) (f : ℝ → ℝ) :
    ⟪ρ.M, σ.M.cfc f⟫ = ∑ i, f (σ.M.H.eigenvalues i) * eigenWeight ρ σ i
```

lemma eigenWeight_nonneg [source]

```
private lemma eigenWeight_nonneg (ρ σ : MState d) (i : d) : 0 ≤ eigenWeight ρ σ i
```

lemma eigenWeight_zero_of_eigenvalue_zero [source]

```
private lemma eigenWeight_zero_of_eigenvalue_zero {i : d} (hσ : σ.M.ker ≤ ρ.M.ker)
  (hei : σ.M.H.eigenvalues i = 0) :
    eigenWeight ρ σ i = 0
```

lemma hasDerivAt_inner_rpow_at_zero [source]

```
private lemma hasDerivAt_inner_rpow_at_zero (h : σ.M.ker ≤ ρ.M.ker) :
    HasDerivAt (fun u : ℝ => ⟪ρ.M, σ.M ^ u⟫) ⟪ρ.M, σ.M.log⟫ 0
```

lemma hasDerivAt_trace_conj_at_one [source]

```
private lemma hasDerivAt_trace_conj_at_one {ρ σ : MState d}
    (h : σ.M.ker ≤ ρ.M.ker) :
    HasDerivAt
      (fun α : ℝ => ((ρ.M.conj (σ.M ^ ((1 - α) / (2 * α))).mat)).trace)
      (-⟪ρ.M, σ.M.log⟫)
      1
```

lemma scalar_rpow_cross_term [source]

```
private lemma scalar_rpow_cross_term {b : ℝ → ℝ} {c : ℝ}
    (hb : HasDerivAt b (deriv b 1) 1) (hc : b 1 = c) (hc_pos : 0 < c) :
    HasDerivAt (fun α => b α ^ α - b α) (c * Real.log c) 1
```

lemma hasDerivAt_trace_rpow_sub_trace [source]

For a PSD matrix A, Tr[A^s] - Tr[A] has derivative ⟪A, log A⟫ at s = 1. This generalizes `hasDerivAt_trace_rpow_at_one` to give the derivative of the difference Tr[A^s] - Tr[A], which equals ⟪A, log A⟫ since d/ds Tr[A] = 0.

```
private lemma hasDerivAt_trace_rpow_sub_trace (A : HermitianMat d ℂ) (hA : 0 ≤ A) :
    HasDerivAt (fun s : ℝ => (A ^ s).trace - A.trace) ⟪A, A.log⟫ 1
```

abbrev B_of [source]

```
private abbrev B_of (ρ σ : MState d) (α : ℝ) : HermitianMat d ℂ
```

lemma B_of_one [source]

```
private lemma B_of_one (ρ σ : MState d) : B_of ρ σ 1 = ρ.M
```

lemma B_of_nonneg [source]

```
private lemma B_of_nonneg (ρ σ : MState d) (α : ℝ) : 0 ≤ B_of ρ σ α
```

lemma trace_rpow_sub_trace_at_one [source]

```
private lemma trace_rpow_sub_trace_at_one (M : HermitianMat d ℂ) :
    (M ^ (1 : ℝ)).trace - M.trace = 0
```

lemma cross_term_at_one [source]

```
private lemma cross_term_at_one (ρ σ : MState d) :
    ((B_of ρ σ 1) ^ (1 : ℝ)).trace - (B_of ρ σ 1).trace
    - (ρ.M ^ (1 : ℝ)).trace + 1 = 0
```

lemma scalar_rpow_cross_term_of_continuous [source]

```
private lemma scalar_rpow_cross_term_of_continuous {b : ℝ → ℝ} {c : ℝ}
    (hb_cont : ContinuousAt b 1) (hc : b 1 = c) (hc_pos : 0 < c)
    (hb_pos : ∀ᶠ α in nhds 1, 0 < b α) :
    HasDerivAt (fun α => b α ^ α - b α) (c * Real.log c) 1
```

lemma scalar_rpow_cross_term_of_continuous_zero [source]

```
private lemma scalar_rpow_cross_term_of_continuous_zero {b : ℝ → ℝ}
    (hb_cont : ContinuousAt b 1) (hc : b 1 = 0)
    (hb_nonneg : ∀ᶠ α in nhds 1, 0 ≤ b α) :
    HasDerivAt (fun α => b α ^ α - b α) 0 1
```

lemma conj_supportProj_eq_of_ker_le [source]

If ker A ≤ ker ρM, then conjugating ρM by the support projection of A gives back ρM. This is because ρM is supported entirely on the support (= range) of A.

```
private lemma conj_supportProj_eq_of_ker_le (A ρM : HermitianMat d ℂ) (hker : A.ker ≤ ρM.ker) :
    ρM.conj (A.supportProj).mat = ρM
```

lemma rpow_tendsto_supportProj [source]

For a PSD matrix A, the function $r \mapsto A \hat{} r$ converges to A.supportProj as $r \to 0$ through nonzero values. On positive eigenvalues λ, $\lambda \hat{} r \to 1$. On zero eigenvalues, $0 \hat{} r = 0$ for $r \neq 0$. So the limit is the support projection (indicator of nonzero eigenvalues).

```
private lemma rpow_tendsto_supportProj
    (A : HermitianMat d ℂ)  :
    Filter.Tendsto (fun r : ℝ => A ^ r) (nhdsWithin 0 {(0 : ℝ)}ᶜ) (nhds A.supportProj)
```

lemma conj_rpow_continuousAt_zero [source]

For PSD matrices A, ρ with $A.ker \leq \rho.ker$, the function $r \mapsto \rho.conj\ (A \hat{} r).mat$ is continuous at $r = 0$. Even though $A \hat{} r$ is discontinuous at $r = 0$ when A has zero eigenvalues, the kernel condition ensures the conj "kills" the discontinuity.

```
private lemma conj_rpow_continuousAt_zero
    (A ρM : HermitianMat d ℂ)
    (hker : A.ker ≤ ρM.ker) :
    ContinuousAt (fun r : ℝ => ρM.conj (A ^ r).mat) 0
```

lemma B_of_continuousAt [source]

```
private lemma B_of_continuousAt (ρ σ : MState d) (h : σ.M.ker ≤ ρ.M.ker) :
    ContinuousAt (B_of ρ σ) 1
```

lemma trace_cfc_sub_le [source]

```
private lemma trace_cfc_sub_le (A : HermitianMat d ℂ) (f g : ℝ → ℝ) :
    |(A.cfc f).trace - (A.cfc g).trace| ≤
      (Fintype.card d : ℝ) * (⨆ i, |f (A.H.eigenvalues i) - g (A.H.eigenvalues i)|)
```

lemma eigenvalues_bounded_near [source]

Eigenvalues of $M(\alpha)$ are uniformly bounded near $\alpha = 1$.

```
private lemma eigenvalues_bounded_near {M : ℝ → HermitianMat d ℂ}
    (hM_nonneg : ∀ᶠ α in nhds 1, 0 ≤ M α)
    (hM_cont : ContinuousAt M 1) :
    ∃ K > 0, ∀ᶠ α in nhds 1, ∀ i, 0 ≤ (M α).H.eigenvalues i ∧ (M α).H.eigenvalues i ≤ K
```

lemma rpow_slope_tendsto_uniformly [source]

```
private lemma rpow_slope_tendsto_uniformly (K : ℝ) :
    ∀ ε > 0, ∃ δ > 0, ∀ h : ℝ, 0 < |h| → |h| < δ →
    ∀ x ∈ Set.Icc 0 K, |(x ^ (1 + h) - x) / h - x * Real.log x| < ε
```

lemma trace_cfc_tendsto_of_tendsto [source]

```
private lemma trace_cfc_tendsto_of_tendsto (f : ℝ → ℝ)
    (hf : ContinuousOn f (Set.Ici 0))
    {M : ℝ → HermitianMat d ℂ}
    (hM_cont : ContinuousAt M 1) (hM_nonneg : ∀ᶠ α in nhds 1, 0 ≤ M α)
    (hM_one : M 1 = ρ.M) :
    Filter.Tendsto (fun α => ((M α).cfc f).trace) (nhds 1) (nhds ((ρ.M.cfc f).trace))
```

lemma cross_term_slope_tendsto_zero [source]

```
private lemma cross_term_slope_tendsto_zero
    {M : ℝ → HermitianMat d ℂ}
    (hM_nonneg : ∀ᶠ α in nhds 1, 0 ≤ M α)
    (hM_cont : ContinuousAt M 1)
    (hM_one : M 1 = ρ.M) :
    Filter.Tendsto
      (fun h : ℝ => ((M (1 + h) ^ (1 + h)).trace - (M (1 + h)).trace
                    - (ρ.M ^ (1 + h)).trace + ρ.M.trace) / h)
      (nhdsWithin 0 {0}ᶜ)
      (nhds 0)
```

lemma hasDerivAt_trace_rpow_sub_trace_variable_base [source]

For a differentiable family of PSD matrices M(α) with M(1) having eigenvalues p_i, the function α ↦ Tr[M(α)^α] - Tr[M(α)] has derivative ⟪M(1), M(1).log⟫ at α = 1. This is because at α = 1, the function x^s - x has zero x-derivative (since d/dx(x^1) = 1), so only the s-derivative contributes, giving the same answer as for fixed eigenvalues.

```
private lemma hasDerivAt_trace_rpow_sub_trace_variable_base
    {M : ℝ → HermitianMat d ℂ}
    (hM_nonneg : ∀ᶠ α in nhds 1, 0 ≤ M α)
    (hM_cont : ContinuousAt M 1)
    (hM_one : M 1 = ρ.M) :
    HasDerivAt (fun α : ℝ => (M α ^ α).trace - (M α).trace) ⟪ρ.M, ρ.M.log⟫ 1
```

lemma rpow_trace_cross_term_vanishes [source]

The cross term in the derivative decomposition vanishes: the function α ↦ Tr[B(α)^α] - Tr[B(α)] - Tr[ρ^α] + 1 has derivative 0 at α = 1. This is because at α=1, B^1 = B, so ∂/∂B Tr[B^α] = Tr[·] (the trace is linear), making the cross term (variation in B times variation in α) vanish.

```
private lemma rpow_trace_cross_term_vanishes {ρ σ : MState d}
    (h : σ.M.ker ≤ ρ.M.ker) :
    HasDerivAt
      (fun α : ℝ => ((ρ.M.conj (σ.M ^ ((1 - α) / (2 * α)))).mat) ^ α).trace
        - (ρ.M.conj (σ.M ^ ((1 - α) / (2 * α)))).mat).trace
        - (ρ.M ^ α).trace + 1)
      0
      1
```

theorem hasDerivAt_trace_at_one [source]

```
private theorem sandwichedRelRentropy.hasDerivAt_trace_at_one {ρ σ : MState d}
    (h : σ.M.ker ≤ ρ.M.ker) :
    HasDerivAt
      (fun α : ℝ => ((ρ.M.conj (σ.M ^ ((1 - α) / (2 * α))).mat) ^ α).trace)
      ⟪ρ.M, ρ.M.log - σ.M.log⟫
      1
```

theorem limit_at_one [source]

The key limit: as $\alpha \to 1$, log(Tr[(ρ.conj σ^t)^α]) / (α-1) → ⟪ρ, log ρ - log σ⟫, where $t = (1-\alpha)/(2\alpha)$. Derived from hasDerivAt_trace_at_one via L'Hôpital (or equivalently, log(1+x)/x → 1 and (f(α)-1)/(α-1) → f'(1)).

```
private theorem sandwichedRelRentropy.limit_at_one (ρ σ : MState d)
    (h : σ.M.ker ≤ ρ.M.ker) :
    Filter.Tendsto
      (fun α : ℝ ↦ ((ρ.M.conj (σ.M ^ ((1 - α) / (2 * α))).mat) ^ α).trace.log / (α - 1))
      (nhdsWithin 1 (Set.Ioi 0 \ {1}))
      (nhds ⟪ρ.M, ρ.M.log - σ.M.log⟫)
```

theorem inner_log_sub_log_nonneg [source]

```
theorem inner_log_sub_log_nonneg (h : σ.M.ker ≤ ρ.M.ker) :
    0 ≤ ⟪ρ.M, ρ.M.log - σ.M.log⟫
```

theorem sandwichedRelRentropy_nonneg [source]

```
theorem sandwichedRelRentropy_nonneg {α : ℝ} (hα : 0 < α) (h : σ.M.ker ≤ ρ.M.ker) :
    0 ≤ if α = 1 then ⟪ρ.M, ρ.M.log - σ.M.log⟫
      else ((ρ.M.conj (σ.M ^ ((1 - α)/(2 * α)) ).mat) ^ α).trace.log / (α - 1)
```

lemma ker_kron_le_of_le [source]

If the kernels of the components are contained, then the kernel of the Kronecker product is contained.

```
lemma ker_kron_le_of_le {d₁ d₂ : Type*} [Fintype d₁] [Fintype d₂] [DecidableEq d₁] [DecidableEq
    d₂]
    (A C : Matrix d₁ d₁ ℂ) (B D : Matrix d₂ d₂ ℂ)
    (hA : LinearMap.ker A.toEuclideanLin ≤ LinearMap.ker C.toEuclideanLin)
    (hB : LinearMap.ker B.toEuclideanLin ≤ LinearMap.ker D.toEuclideanLin) :
    LinearMap.ker (A.kronecker B).toEuclideanLin ≤ LinearMap.ker (C.kronecker D).toEuclideanLin
```

lemma ker_le_of_ker_kron_le_left [source]

If the kernel of a product state is contained in another, the left component kernel is contained.

```
lemma ker_le_of_ker_kron_le_left (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂)
  (h : (σ₁ ⊗ᴹ σ₂).M.ker ≤ (ρ₁ ⊗ᴹ ρ₂).M.ker) :
    σ₁.M.ker ≤ ρ₁.M.ker
```

lemma ker_le_of_ker_kron_le_right [source]

If the kernel of a product state is contained in another, the right component kernel is contained.

```
lemma ker_le_of_ker_kron_le_right (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂)
  (h : (σ₁ ⊗ᴹ σ₂).M.ker ≤ (ρ₁ ⊗ᴹ ρ₂).M.ker) :
    σ₂.M.ker ≤ ρ₂.M.ker
```

lemma ker_prod_le_iff [source]

The kernel of a product state is contained in another product state's kernel iff the individual kernels are contained.

```
lemma ker_prod_le_iff (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂) :
    (σ₁ ⊗ᴹ σ₂).M.ker ≤ (ρ₁ ⊗ᴹ ρ₂).M.ker ↔ σ₁.M.ker ≤ ρ₁.M.ker ∧ σ₂.M.ker ≤ ρ₂.M.ker
```

lemma inner_kron [source]

```
lemma HermitianMat.inner_kron
    (A : HermitianMat d₁ ℂ) (B : HermitianMat d₂ ℂ) (C : HermitianMat d₁ ℂ) (D : HermitianMat d
    ₂ ℂ) :
    ⟪A ⊗ₖ B, C ⊗ₖ D⟫ = ⟪A, C⟫ * ⟪B, D⟫
```

lemma continuousOn_rpow_uniform [source]

```
lemma continuousOn_rpow_uniform {K : Set ℝ} (hK : IsCompact K) :
    ContinuousOn (fun r : ℝ ↦ UniformOnFun.ofFun {K} (fun t : ℝ ↦ t ^ r)) (Set.Ioi 0)
```

theorem sandwichedRelRentropy_additive_alpha_one_aux [source]

```
theorem sandwichedRelRentropy_additive_alpha_one_aux (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂)
  (h1 : σ₁.M.ker ≤ ρ₁.M.ker) (h2 : σ₂.M.ker ≤ ρ₂.M.ker) :
    ⟪(ρ₁ ⊗ᴹ ρ₂).M, (ρ₁ ⊗ᴹ ρ₂).M.log - (σ₁ ⊗ᴹ σ₂).M.log⟫ =
    ⟪ρ₁.M, ρ₁.M.log - σ₁.M.log⟫_ℝ + ⟪ρ₂.M, ρ₂.M.log - σ₂.M.log⟫
```

def SandwichedRelRentropy [source]

The Sandwiched Renyi Relative Entropy, defined with ln (nits). Note that at `α = 1` this definition switch to the standard Relative Entropy, for continuity. For $\alpha \leq 0$, this gives junk value 0. (There is no conventional value for $\alpha < 0$; there is a continuous limit at $\alpha = 0$, but it is complicated and unneeded at the moment.)

```
def SandwichedRelRentropy (α : ℝ) (ρ σ : MState d) : ENNReal
```

def qRelativeEnt [source]

The quantum relative entropy $\mathbf{D}(\rho\|\sigma) :=$ `Tr[ρ (log ρ - log σ)]`. Also called the Umegaki quantum relative entropy, when it's necessary to distinguish from other relative entropies.

```
def qRelativeEnt (ρ σ : MState d) : ENNReal
```

theorem sandwichedRelRentropy_additive_alpha_one [source]

The Sandwiched Renyi Relative entropy is additive for α=1 (standard relative entropy).

```
private theorem sandwichedRelRentropy_additive_alpha_one (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d
    ₂) :
    D̃ 1(ρ₁ ⊗ᴹ ρ₂‖σ₁ ⊗ᴹ σ₂) = D̃ 1(ρ₁‖σ₁) + D̃ 1(ρ₂‖σ₂)
```

lemma sandwiched_term_product [source]

```
lemma sandwiched_term_product (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂) (α β : ℝ) :
    (((ρ₁ ⊗ᴹ ρ₂).M.conj ((σ₁ ⊗ᴹ σ₂).M ^ β).mat) ^ α).trace =
    ((ρ₁.M.conj (σ₁.M ^ β).mat) ^ α).trace * ((ρ₂.M.conj (σ₂.M ^ β).mat) ^ α).trace
```

theorem sandwichedRelRentropy_additive_alpha_ne_one [source]

```
theorem sandwichedRelRentropy_additive_alpha_ne_one {α : ℝ} (hα : α ≠ 1) (ρ₁ σ₁ : MState d₁) (
    ρ₂ σ₂ : MState d₂) :
    D̃ α(ρ₁ ⊗ᴹ ρ₂‖σ₁ ⊗ᴹ σ₂) = D̃ α(ρ₁‖σ₁) + D̃ α(ρ₂‖σ₂)
```

theorem sandwichedRelRentropy_additive [source]

The Sandwiched Renyi Relative entropy is additive when the inputs are product states

```
@[simp]
theorem sandwichedRelRentropy_additive (α) (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂) :
    D̃ α(ρ₁ ⊗ᴹ ρ₂‖σ₁ ⊗ᴹ σ₂) = D̃ α(ρ₁‖σ₁) + D̃ α(ρ₂‖σ₂)
```

theorem qRelativeEnt_additive [source]

The quantum relative entropy is additive when the inputs are product states

```
@[simp]
theorem qRelativeEnt_additive (ρ₁ σ₁ : MState d₁) (ρ₂ σ₂ : MState d₂) :
    𝐃(ρ₁ ⊗ᴹ ρ₂‖σ₁ ⊗ᴹ σ₂) = 𝐃(ρ₁‖σ₁) + 𝐃(ρ₂‖σ₂)
```

theorem sandwichedRelRentropy_relabel [source]

```
@[simp]
theorem sandwichedRelRentropy_relabel (ρ σ : MState d) (e : d₂ ≃ d) :
    D̃ α(ρ.relabel e‖σ.relabel e) = D̃ α(ρ‖σ)
```

theorem sandwichedRelRentropy_self [source]

```
@[simp]
theorem sandwichedRelRentropy_self (hα : 0 < α) (ρ : MState d) :
```

theorem sandwichedRelEntropy_ne_top [source]

```
@[aesop (rule_sets := [finiteness]) unsafe apply]
theorem sandwichedRelEntropy_ne_top {ρ σ : MState d} [σ.M.NonSingular] : D̃ α(ρ‖σ) ≠ ⊤
```

lemma continuousOn_exponent [source]

```
@[fun_prop]
lemma continuousOn_exponent : ContinuousOn (fun α : ℝ => (1 - α) / (2 * α)) (Set.Ioi 0)
```

lemma continuousOn_cpow_const_Ioi [source]

```
@[fun_prop]
lemma Complex.continuousOn_cpow_const_Ioi (z : ℂ) :
    ContinuousOn (fun r : ℝ => z ^ (r : ℂ)) (Set.Ioi 0)
```

lemma maps_to_Iio_of_Ioi_1 [source]

The function $\alpha \mapsto (1 - \alpha) / (2 * \alpha)$ maps the interval $(1, \infty)$ to $(-\infty, 0)$.

```
lemma maps_to_Iio_of_Ioi_1 : Set.MapsTo (fun α : ℝ => (1 - α) / (2 * α)) (Set.Ioi 1) (Set.Iio
    0)
```

theorem frontier_singleton [source]

```
@[simp]
theorem frontier_singleton {X : Type*} [TopologicalSpace X] [T1Space X] [PerfectSpace X]
    (p : X) : frontier {p} = {p}
```

theorem continuousOn_Ioi_1_aux [source]

```
private theorem sandwichedRelRentropy.continuousOn_Ioi_1_aux (ρ σ : MState d) :
    ContinuousOn (fun (α : ℝ) ↦ ((HermitianMat.conj (σ.M ^ ((1 - α) / (2 * α))).mat) ρ.M ^ α))
    (Set.Ioi 1)
```

theorem continuousOn_Ioi_1 [source]

```
private theorem sandwichedRelRentropy.continuousOn_Ioi_1 (ρ σ : MState d) :
    ContinuousOn (fun α => D̃ α(ρ‖σ)) (Set.Ioi 1)
```

theorem continuousOn_Ioo_0_1_aux [source]

```
private theorem sandwichedRelRentropy.continuousOn_Ioo_0_1_aux (ρ σ : MState d) :
    ContinuousOn (fun (α : ℝ) ↦ ((HermitianMat.conj (σ.M ^ ((1 - α) / (2 * α))).mat) ρ.M ^ α))
    (Set.Ioo 0 1)
```

theorem continuousOn_Ioo_0_1 [source]

Continuity on (0,1): the sandwich relative Rényi entropy is continuous in α on (0,1).

```
private theorem sandwichedRelRentropy.continuousOn_Ioo_0_1 (ρ σ : MState d) :
    ContinuousOn (fun α => D̃ α(ρ‖σ)) (Set.Ioo 0 1)
```

theorem continuousAt_1 [source]

Continuity at 1: the sandwich relative Rényi entropy is continuous at $\alpha = 1$.

```
private theorem sandwichedRelRentropy.continuousAt_1 (ρ σ : MState d) :
    ContinuousWithinAt (fun α => D̃ α(ρ‖σ)) (Set.Ioi 0) 1
```

theorem continuousOn [source]

```
@[fun_prop]
theorem sandwichedRelRentropy.continuousOn (ρ σ : MState d) :
    ContinuousOn (fun α => D̃ α(ρ‖σ)) (Set.Ioi 0)
```

theorem qRelativeEnt_ker [source]

Quantum relative entropy as `Tr[ρ (log ρ - log σ)]` when supports are contained.

```
theorem qRelativeEnt_ker {ρ σ : MState d} (h : σ.M.ker ≤ ρ.M.ker) :
    𝐃(ρ‖σ).toEReal = ⟪ρ.M, ρ.M.log - σ.M.log⟫
```

theorem qRelativeEnt_eq_neg_S$_{vn}$_add [source]

```
theorem qRelativeEnt_eq_neg_S$_{vn}$_add (ρ σ : MState d) :
    (qRelativeEnt ρ σ).toEReal = -(S$_{vn}$ ρ : EReal) +
      if σ.M.ker ≤ ρ.M.ker then (-⟪ρ.M, σ.M.log⟫ : EReal) else (⊤ : EReal)
```

theorem qRelativeEnt_relabel [source]

The quantum relative entropy is unchanged by `MState.relabel`

```
@[simp]
theorem qRelativeEnt_relabel (ρ σ : MState d) (e : d₂ ≃ d) :
    𝐃(ρ.relabel e‖σ.relabel e) = 𝐃(ρ‖σ)
```

theorem sandwichedRelRentropy_of_unique [source]

```
@[simp]
theorem sandwichedRelRentropy_of_unique [Unique d] (ρ σ : MState d) :
    D̃ α(ρ‖σ) = 0
```

theorem qRelEntropy_of_unique [source]

```
@[simp]
theorem qRelEntropy_of_unique [Unique d] (ρ σ : MState d) :
    𝐃(ρ‖σ) = 0
```

theorem sandwichedRelRentropy_heq_congr [source]

```
theorem sandwichedRelRentropy_heq_congr
      {d₁ d₂ : Type u} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq d₂]
      {ρ₁ σ₁ : MState d₁} {ρ₂ σ₂ : MState d₂} (hd : d₁ = d₂) (hρ : ρ₁ ≍ ρ₂) (hσ : σ₁ ≍ σ₂) :
    D̃ α(ρ₁‖σ₁) = D̃ α(ρ₂‖σ₂)
```

theorem sandwichedRelRentropy_congr [source]

```
@[gcongr]
theorem sandwichedRelRentropy_congr {α : ℝ}
      {d₁ d₂ : Type u} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq d₂]
      {ρ₁ σ₁ : MState d₁} {ρ₂ σ₂ : MState d₂} (hd : d₁ = d₂)
        (hρ : ρ₁ = ρ₂.relabel (Equiv.cast hd)) (hσ : σ₁ = σ₂.relabel (Equiv.cast hd)) :
    D̃ α(ρ₁‖σ₁) = D̃ α(ρ₂‖σ₂)
```

theorem qRelEntropy_heq_congr [source]

```
@[gcongr]
theorem qRelEntropy_heq_congr {d₁ d₂ : Type u} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [
    DecidableEq d₂]
      {ρ₁ σ₁ : MState d₁} {ρ₂ σ₂ : MState d₂} (hd : d₁ = d₂) (hρ : ρ₁ ≍ ρ₂) (hσ : σ₁ ≍ σ₂) :
    𝐃(ρ₁‖σ₁) = 𝐃(ρ₂‖σ₂)
```

theorem qRelativeEnt_rank [source]

Quantum relative entropy when σ has full rank

```
theorem qRelativeEnt_rank {ρ σ : MState d} [σ.M.NonSingular] :
    (𝐃(ρ‖σ) : EReal) = ⟪ρ.M, ρ.M.log - σ.M.log⟫
```

def approxLog [source]

```
private def approxLog (N : ℕ) : ℝ → ℝ
```

lemma approxLog_continuous [source]

```
private lemma approxLog_continuous (N : ℕ) : Continuous (approxLog N)
```

lemma approxLog_ge_log_pos [source]

```
private lemma approxLog_ge_log_pos {t : ℝ} (ht : 0 < t) (N : ℕ) :
    Real.log t ≤ approxLog N t
```

lemma continuous_inner_cfc_approxLog [source]

```
private lemma continuous_inner_cfc_approxLog (ρ : MState d) (N : ℕ) :
    Continuous (fun σ : MState d => ⟪ρ.M, σ.M.cfc (approxLog N)⟫)
```

lemma approxLog_tendsto_at_pos [source]

```
private lemma approxLog_tendsto_at_pos {t : ℝ} (ht : 0 < t) :
    Filter.Tendsto (fun N : ℕ => approxLog N t) Filter.atTop (nhds (Real.log t))
```

lemma inner_cfc_approxLog_ge [source]

```
private lemma inner_cfc_approxLog_ge (ρ σ : MState d) (N : ℕ) (hσ : σ.M.ker ≤ ρ.M.ker) :
    ⟪ρ.M, σ.M.log⟫ ≤ ⟪ρ.M, σ.M.cfc (approxLog N)⟫
```

lemma tendsto_inner_cfc_approxLog [source]

```
private lemma tendsto_inner_cfc_approxLog (ρ x : MState d) (hx : x.M.ker ≤ ρ.M.ker) :
    Filter.Tendsto (fun N : ℕ => ⟪ρ.M, x.M.cfc (approxLog N)⟫)
      Filter.atTop (nhds ⟪ρ.M, x.M.log⟫)
```

lemma inner_log_bounded_near [source]

```
lemma inner_log_bounded_near (hx : σ.M.ker ≤ ρ.M.ker) {y : ℝ} (hy : ⟪ρ.M, σ.M.log⟫ < y) :
    ∀ᶠ x in nhds σ, x.M.ker ≤ ρ.M.ker → ⟪ρ.M, x.M.log⟫ < y
```

lemma eigenWeight_eq_zero_iff [source]

```
private lemma eigenWeight_eq_zero_iff (ρ x : MState d) (i : d) :
    eigenWeight ρ x i = 0 ↔ (x.M.H.eigenvectorBasis i : EuclideanSpace ℂ d) ∈ ρ.M.ker
```

lemma ker_le_iff_eigenWeight_zero [source]

```
private lemma ker_le_iff_eigenWeight_zero (ρ x : MState d) :
    x.M.ker ≤ ρ.M.ker ↔ ∀ i, x.M.H.eigenvalues i = 0 → eigenWeight ρ x i = 0
```

lemma neg_ker_exists_eigenWeight_pos [source]

```
private lemma neg_ker_exists_eigenWeight_pos (ρ x : MState d) (hx : ¬(x.M.ker ≤ ρ.M.ker)) :
    ∃ i, x.M.H.eigenvalues i = 0 ∧ 0 < eigenWeight ρ x i
```

lemma approxLog_at_zero [source]

```
private lemma approxLog_at_zero (N : ℕ) : approxLog N 0 = -(N : ℝ)
```

lemma inner_cfc_approxLog_tendsto_bot [source]

```
private lemma inner_cfc_approxLog_tendsto_bot (ρ x : MState d) (hx : ¬(x.M.ker ≤ ρ.M.ker)) :
    Filter.Tendsto (fun N : ℕ => ⟪ρ.M, x.M.cfc (approxLog N)⟫) Filter.atTop Filter.atBot
```

theorem qRelativeEnt_lowerSemicontinuous_2 [source]

```
theorem qRelativeEnt_lowerSemicontinuous_2 (ρ x : MState d) (hx : ¬(x.M.ker ≤ ρ.M.ker)) (y :
    ENNReal) (hy : y < ⊤) :
    ∀ᶠ (x' : MState d) in nhds x,
      y < (if x'.M.ker ≤ ρ.M.ker then ⟪ρ.M, ρ.M.log - x'.M.log⟫ else ⊤ : EReal)
```

theorem lowerSemicontinuous [source]

```
@[fun_prop]
theorem qRelativeEnt.lowerSemicontinuous (ρ : MState d) : LowerSemicontinuous fun σ => 𝐃(ρ‖σ)
```

theorem qRelativeEnt_joint_convexity [source]

Joint convexity of Quantum relative entropy. We can't state this with `ConvexOn` because that requires an `AddCommMonoid`, which `MStates` are not. Instead we state it with `Mixable`.

*TODO: * Add the `Mixable` instance that infers from the `Coe` so that the right hand side can be written as `p [D(ρ₁‖σ₁) ↔ D(ρ₂‖σ₂)]` * Define (joint) convexity as its own thing - a `ConvexOn` for `Mixable` types. * Maybe, more broadly, find a way to make `ConvexOn` work with the subset of `Matrix` that corresponds to `MState`.*

```
theorem qRelativeEnt_joint_convexity :
  ∀ (ρ₁ ρ₂ σ₁ σ₂ : MState d), ∀ (p : Prob),
    D(p [ρ₁ ↔ ρ₂]‖p [σ₁ ↔ σ₂]) ≤ p * D(ρ₁‖σ₁) + (1 - p) * D(ρ₂‖σ₂)
```

theorem qRelEntropy_self [source]

```
@[simp]
theorem qRelEntropy_self (ρ : MState d) : D(ρ‖ρ) = 0
```

theorem qRelativeEnt_ne_top [source]

```
@[aesop (rule_sets := [finiteness]) unsafe apply]
theorem qRelativeEnt_ne_top {ρ σ : MState d} [σ.M.NonSingular] : D(ρ‖σ) ≠ ⊤
```

theorem qMutualInfo_as_qRelativeEnt [source]

$$I(A:B) = \mathbf{D}(\rho^{AB} \| \rho^A \otimes \rho^B)$$

```
theorem qMutualInfo_as_qRelativeEnt (ρ : MState (dA × dB)) :
    qMutualInfo ρ = (D(ρ‖ρ.traceRight ⊗ᴹ ρ.traceLeft) : EReal)
```

lemma pos_of_MState_le_smul [source]

```
private lemma pos_of_MState_le_smul {σ₁ σ₂ : MState d} (hσ : σ₂.M ≤ α • σ₁.M) : 0 < α
```

lemma inner_log_mono_of_posDef_of_le [source]

```
private lemma HermitianMat.inner_log_mono_of_posDef_of_le {A B C : HermitianMat d 𝕜}
    (hC : 0 ≤ C) (hA : A.mat.PosDef) (hAB : A ≤ B) :
    ⟪C, A.log⟫ ≤ ⟪C, B.log⟫
```

lemma posDef_add_eps [source]

```
private lemma posDef_add_eps {A : HermitianMat d ℂ} (hA : 0 ≤ A) {ε : ℝ} (hε : 0 < ε) :
    (A + ε • 1).mat.PosDef
```

lemma log_add_eps_eq_cfc [source]

```
private lemma log_add_eps_eq_cfc (A : HermitianMat d 𝕜) (ε : ℝ) :
    (A + ε • 1).log = A.cfc (Real.log <| · + ε)
```

lemma inner_cfc_eq_sum [source]

```
private lemma inner_cfc_eq_sum (A C : HermitianMat d 𝕜) (f : ℝ → ℝ) :
    ⟪C, A.cfc f⟫ = ∑ i, f (A.H.eigenvalues i) *
      RCLike.re ((C.mat * (A.H.eigenvectorUnitary.val * Matrix.single i i 1 * A.H.
    eigenvectorUnitary.val.conjTranspose)).trace)
```

lemma eigenproj_coeff_zero_of_ker_le [source]

```
private lemma eigenproj_coeff_zero_of_ker_le {A C : HermitianMat d 𝕜}
    (hker : A.ker ≤ C.ker) {i : d} (hi : A.H.eigenvalues i = 0) :
    RCLike.re ((C.mat * (A.H.eigenvectorUnitary.val * Matrix.single i i 1 * A.H.
    eigenvectorUnitary.val.conjTranspose)).trace) = 0
```

lemma inner_log_shift_tendsto [source]

```
private lemma inner_log_shift_tendsto {A C : HermitianMat d 𝕜} (hker : A.ker ≤ C.ker) :
    (𝓝[>] 0).Tendsto (fun (ε : ℝ) ↦ ⟪C, (A + ε • 1).log⟫) (𝓝 ⟪C, A.log⟫)
```

lemma inner_log_mono_of_psd_of_le [source]

```
private lemma HermitianMat.inner_log_mono_of_psd_of_le {A B C : HermitianMat d ℂ}
    (hA : 0 ≤ A) (hAB : A ≤ B) (hC : 0 ≤ C) (hker : A.ker ≤ C.ker) :
    ⟪C, A.log⟫ ≤ ⟪C, B.log⟫
```

lemma inner_log_sub_le_log_alpha [source]

```
private lemma HermitianMat.inner_log_sub_le_log_alpha (ρ : MState d) {σ₁ σ₂ : MState d} {α : ℝ}
    (hσ : σ₂.M ≤ α • σ₁.M)
    (hker₁ : σ₁.M.ker ≤ ρ.M.ker) (hker₂ : σ₂.M.ker ≤ ρ.M.ker) :
    ⟪ρ.M, σ₂.M.log - σ₁.M.log⟫ ≤ Real.log α
```

theorem qRelEntropy_le_add_of_le_smul [source]

```
theorem qRelEntropy_le_add_of_le_smul (ρ : MState d) {σ₁ σ₂ : MState d} (hσ : σ₂.M ≤ α • σ₁.M)
    :
    𝐃(ρ‖σ₁) ≤ 𝐃(ρ‖σ₂) + ENNReal.ofReal (Real.log α)
```

theorem qRelativeEnt_op_le [source]

```
theorem qRelativeEnt_op_le {ρ σ : MState d} (h : ρ.M ≤ α • σ.M) :
    𝐃(ρ‖σ) ≤ .ofReal (Real.log α)
```

2.41 QuantumInfo.Finite.Entropy.SSA

QuantumInfo/Finite/Entropy/SSA.lean on GitHub.

def opNorm [source]

The operator norm of a matrix, with respect to the Euclidean norm (l2 norm) on the domain and codomain.

```
noncomputable def Matrix.opNorm [RCLike 𝕜] (A : Matrix m n 𝕜) : ℝ
```

theorem isometry_preserves_norm [source]

```
theorem Matrix.isometry_preserves_norm (A : Matrix n m 𝕜) (hA : A.Isometry) (x : EuclideanSpace
    𝕜 m) :
  ‖Matrix.toEuclideanLin A x‖ = ‖x‖
```

theorem opNorm_isometry [source]

```
theorem Matrix.opNorm_isometry [Nonempty m] (A : Matrix n m 𝕜) (hA : A.Isometry) : Matrix.
  opNorm A = 1
```

def map_to_tensor_MES [source]

The matrix representation of the map $v \{}mapsto v \{}otimes \{}sum_k /k\{}rangle/k\{}rangle$. The output index is `(d1 \{}times d2) \{}times d2`.

```
def map_to_tensor_MES : Matrix ((d₁ × d₂) × d₂) d₁ ℂ
```

theorem map_to_tensor_MES_prop [source]

```
theorem map_to_tensor_MES_prop (A : Matrix (d₁ × d₂) (d₁ × d₂) ℂ) :
  (map_to_tensor_MES d₁ d₂).conjTranspose * (Matrix.kronecker A (1 : Matrix d₂ d₂ ℂ)) * (
  map_to_tensor_MES d₁ d₂) =
  A.traceRight
```

def V_rho [source]

The isometry V_rho from the paper.

```
noncomputable def V_rho (ρAB : HermitianMat (dA × dB) ℂ) : Matrix ((dA × dB) × dB) dA ℂ
```

def V_sigma [source]

The isometry V_sigma from the paper.

```
noncomputable def V_sigma (σBC : HermitianMat (dB × dC) ℂ) : Matrix (dB × (dB × dC)) dC ℂ
```

lemma V_rho_conj_mul_self_eq [source]

*V_rho^H * V_rho simplifies to sandwiching the traceRight by the inverse square root.*

```
lemma V_rho_conj_mul_self_eq (ρAB : HermitianMat (dA × dB) ℂ) (hp : ρAB.mat.PosDef) :
  let ρA
```

lemma PosDef_traceRight [source]

The partial trace (left) of a positive definite matrix is positive definite.

```
lemma PosDef_traceRight [Nonempty dB] (A : HermitianMat (dA × dB) ℂ) (hA : A.mat.PosDef) :
  A.traceRight.mat.PosDef
```

lemma PosDef_traceLeft [source]

```
lemma PosDef_traceLeft [Nonempty dA] (A : HermitianMat (dA × dB) ℂ) (hA : A.mat.PosDef) :
    A.traceLeft.mat.PosDef
```

theorem V_rho_isometry [source]

V_rho is an isometry.

```
theorem V_rho_isometry [Nonempty dB] (ρAB : HermitianMat (dA × dB) ℂ) (hρ : ρAB.mat.PosDef) :
    (V_rho ρAB)^H * (V_rho ρAB) = 1
```

theorem V_sigma_isometry [source]

V_sigma is an isometry.

```
theorem V_sigma_isometry [Nonempty dB] (σBC : HermitianMat (dB × dC) ℂ) (hσ : σBC.mat.PosDef) :
    (V_sigma σBC).conjTranspose * (V_sigma σBC) = 1
```

def W_mat [source]

The operator W from the paper, defined as a matrix product.

```
noncomputable def W_mat (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ) :
    Matrix ((dA × dB) × dC) ((dA × dB) × dC) ℂ
```

theorem opNorm_mul_le [source]

The operator norm of a matrix product is at most the product of the operator norms.

```
theorem Matrix.opNorm_mul_le {l m n 𝕜 : Type*} [Fintype l] [Fintype m] [Fintype n]
    [DecidableEq l] [DecidableEq m] [DecidableEq n] [RCLike 𝕜]
    (A : Matrix l m 𝕜) (B : Matrix m n 𝕜) :
    Matrix.opNorm (A * B) ≤ Matrix.opNorm A * Matrix.opNorm B
```

theorem opNorm_reindex_proven [source]

```
theorem Matrix.opNorm_reindex_proven {l m n p : Type*} [Fintype l] [Fintype m] [Fintype n] [
    Fintype p]
    [DecidableEq l] [DecidableEq m] [DecidableEq n] [DecidableEq p]
    (e : m ≃ l) (f : n ≃ p) (A : Matrix m n 𝕜) :
    Matrix.opNorm (A.reindex e f) = Matrix.opNorm A
```

def U_rho [source]

Define U_rho as the Kronecker product of V_rho and the identity.

```
noncomputable def U_rho (ρAB : HermitianMat (dA × dB) ℂ) : Matrix (((dA × dB) × dB) × dC) (dA
    × dC) ℂ
```

def U_sigma [source]

Define U_sigma as the Kronecker product of the identity and V_sigma.

```
noncomputable def U_sigma (σBC : HermitianMat (dB × dC) ℂ) : Matrix (dA × (dB × (dB × dC))) (dA
    × dC) ℂ
```

theorem opNorm_conjTranspose_eq_opNorm [source]

The operator norm of the conjugate transpose is equal to the operator norm.

```
theorem Matrix.opNorm_conjTranspose_eq_opNorm {m n : Type*} [Fintype m] [Fintype n]
    [DecidableEq m] [DecidableEq n] (A : Matrix m n 𝕜) :
    Matrix.opNorm A^H = Matrix.opNorm A
```

theorem isometry_mul_conjTranspose_le_one [source]

```
theorem isometry_mul_conjTranspose_le_one {m n : Type*} [Fintype m] [Fintype n]
    [DecidableEq m] [DecidableEq n]
    (V : Matrix m n ℂ) (hV : V.conjTranspose * V = 1) :
    V * V.conjTranspose ≤ 1
```

theorem conjTranspose_isometry_mul_isometry_le_one [source]

```
theorem conjTranspose_isometry_mul_isometry_le_one {m n k : Type*}
    [Fintype m] [Fintype n] [Fintype k] [DecidableEq m] [DecidableEq n] [DecidableEq k]
    (A : Matrix k m ℂ) (B : Matrix k n ℂ)
    (hA : A.conjTranspose * A = 1) (hB : B.conjTranspose * B = 1) :
    (A.conjTranspose * B).conjTranspose * (A.conjTranspose * B) ≤ 1
```

theorem reindex_le_reindex_iff [source]

```
theorem HermitianMat.reindex_le_reindex_iff {d d₂ : Type*} [Fintype d] [DecidableEq d]
    [Fintype d₂] [DecidableEq d₂] (e : d ≃ d₂) (A B : HermitianMat d ℂ) :
    A.reindex e ≤ B.reindex e ↔ A ≤ B
```

theorem inv_kronecker [source]

```
theorem HermitianMat.inv_kronecker {m n : Type*} [Fintype m] [DecidableEq m]
    [Fintype n] [DecidableEq n] [Nonempty m] [Nonempty n]
    (A : HermitianMat m ℂ) (B : HermitianMat n ℂ)
    [A.NonSingular] [B.NonSingular] :
    (A ⊗ₖ B)⁻¹ = A⁻¹ ⊗ₖ B⁻¹
```

theorem inv_reindex [source]

```
theorem HermitianMat.inv_reindex {d d₂ : Type*} [Fintype d] [DecidableEq d]
    [Fintype d₂] [DecidableEq d₂] (A : HermitianMat d ℂ) (e : d ≃ d₂) :
    (A.reindex e)⁻¹ = A⁻¹.reindex e
```

theorem PosDef_kronecker [source]

```
theorem HermitianMat.PosDef_kronecker {m n : Type*} [Fintype m] [DecidableEq m]
    [Fintype n] [DecidableEq n]
    (A : HermitianMat m ℂ) (B : HermitianMat n ℂ)
    (hA : A.mat.PosDef) (hB : B.mat.PosDef) :
```

```
    (A ⊗ₖ B).mat.PosDef
```

theorem PosDef_reindex [source]

```
theorem HermitianMat.PosDef_reindex {d d₂ : Type*} [Fintype d] [DecidableEq d]
    [Fintype d₂] [DecidableEq d₂] (A : HermitianMat d ℂ) (e : d ≃ d₂)
    (hA : A.mat.PosDef) :
    (A.reindex e).mat.PosDef
```

def S_mat [source]

The sandwich matrix S used in the proof of intermediate_ineq. This is derived from W_mat_sq_le_one by algebraic manipulation (conjugation and simplification).

```
private noncomputable def S_mat (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC)
    ℂ) :
    Matrix ((dA × dB) × dC) ((dA × dB) × dC) ℂ
```

lemma W_mat_sq_eq_conj [source]

```
private lemma W_mat_sq_eq_conj [Nonempty dA] [Nonempty dB] [Nonempty dC]
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (hρ : ρAB.mat.PosDef) (hσ : σBC.mat.PosDef) :
    (W_mat ρAB σBC)^H * (W_mat ρAB σBC) =
      S_mat ρAB σBC * (ρAB ⊗ₖ (σBC.traceLeft)⁻¹).mat * S_mat ρAB σBC
```

lemma S_mat_conj_rhs_eq_one [source]

```
private lemma S_mat_conj_rhs_eq_one [Nonempty dA] [Nonempty dB] [Nonempty dC]
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (hρ : ρAB.mat.PosDef) (hσ : σBC.mat.PosDef) :
    S_mat ρAB σBC * ((ρAB.traceRight ⊗ₖ σBC⁻¹).reindex (Equiv.prodAssoc dA dB dC).symm).mat *
      S_mat ρAB σBC = 1
```

lemma W_mat_eq_factored [source]

```
private lemma W_mat_eq_factored
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ) :
    W_mat ρAB σBC =
      let F
```

lemma F_tensor_MES_eq_V_rho [source]

```
private lemma F_tensor_MES_eq_V_rho
    (ρAB : HermitianMat (dA × dB) ℂ) :
    let F
```

abbrev BigIdx [source]

```
abbrev BigIdx (dA dB dC : Type*)
```

abbrev SmallIdx [source]

```
abbrev SmallIdx (dA dB dC : Type*)
```

abbrev MidIdx [source]

```
abbrev MidIdx (dA dB dC : Type*)
```

def assoc_equiv [source]

The associator equivalence (no swap needed). Maps $(((dA\times dB)\times dB)\times(dB\times dC))$ to $((dA\times dB)\times(dB\times(dB\times dC)))$.

```
private def assoc_equiv (dA dB dC : Type*) :
    BigIdx dA dB dC ≃ MidIdx dA dB dC
```

def T₁_mat [source]

```
private def T₁_mat (ρAB : HermitianMat (dA × dB) ℂ) :
    Matrix (BigIdx dA dB dC) (SmallIdx dA dB dC) ℂ
```

def T₂_mat [source]

```
private def T₂_mat (σBC : HermitianMat (dB × dC) ℂ) :
    Matrix (SmallIdx dA dB dC) (MidIdx dA dB dC) ℂ
```

def PERM_mat [source]

```
private def PERM_mat (dA dB dC : Type*) [Fintype dA] [Fintype dB] [Fintype dC]
    [DecidableEq dA] [DecidableEq dB] [DecidableEq dC] :
    Matrix (MidIdx dA dB dC) (BigIdx dA dB dC) ℂ
```

lemma T₁_isometry [source]

```
private lemma T₁_isometry [Nonempty dB]
    (ρAB : HermitianMat (dA × dB) ℂ) (hρ : ρAB.mat.PosDef) :
    (T₁_mat dA dB dC ρAB)^H * (T₁_mat dA dB dC ρAB) = 1
```

lemma T₂_sq_le_one [source]

```
private lemma T₂_sq_le_one [Nonempty dB]
    (σBC : HermitianMat (dB × dC) ℂ) (hσ : σBC.mat.PosDef) :
    (T₂_mat dA dB dC σBC)^H * (T₂_mat dA dB dC σBC) ≤ 1
```

lemma PERM_isometry [source]

```
private lemma PERM_isometry : (PERM_mat dA dB dC)^H * PERM_mat dA dB dC = 1
```

lemma W_mat_entry [source]

*Element-wise identity: W_mat = ∑_{b*} V_rho * V_sigma†. This is the key computation from Eq. (6) of Lin-Kim-Hsieh.*

```
private lemma W_mat_entry (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (i j : SmallIdx dA dB dC) :
    W_mat ρAB σBC i j =
      ∑ b_star : dB,
        V_rho ρAB ((i.1, b_star)) j.1.1 *
        (V_sigma σBC)^H i.2 (b_star, (j.1.2, j.2))
```

lemma RHS_entry [source]

*Element-wise identity: RHS = ∑_{b*} V_rho * V_sigma†.*

```
private lemma RHS_entry (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (i j : SmallIdx dA dB dC) :
    (T₂_mat dA dB dC σBC * PERM_mat dA dB dC * T₁_mat dA dB dC ρAB) i j =
      ∑ b_star : dB,
        V_rho ρAB ((i.1, b_star)) j.1.1 *
        (V_sigma σBC)^H i.2 (b_star, (j.1.2, j.2))
```

lemma W_mat_eq_three_factors [source]

```
private lemma W_mat_eq_three_factors [Nonempty dA] [Nonempty dB] [Nonempty dC]
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ) :
    W_mat ρAB σBC =
      T₂_mat dA dB dC σBC * PERM_mat dA dB dC * T₁_mat dA dB dC ρAB
```

theorem W_mat_sq_le_one [source]

Core inequality: W†W ≤ I. This is the key step, following from the isometry argument: V_rho ⊗ I_C and I_A ⊗ V_sigma are isometries, their cross product has norm ≤ 1, and the result can be related to W_mat through the MES computation (Eq. 6 in Lin-Kim-Hsieh).

```
theorem W_mat_sq_le_one [Nonempty dA] [Nonempty dB] [Nonempty dC]
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (hρ : ρAB.mat.PosDef) (hσ : σBC.mat.PosDef) :
    (W_mat ρAB σBC)^H * (W_mat ρAB σBC) ≤ 1
```

lemma S_mat_isUnit [source]

```
private lemma S_mat_isUnit [Nonempty dA] [Nonempty dB] [Nonempty dC]
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (hρ : ρAB.mat.PosDef) (hσ : σBC.mat.PosDef) :
    IsUnit (S_mat ρAB σBC)
```

theorem intermediate_ineq [source]

The intermediate operator inequality: ρ_AB ⊗ σ_C^{-1} ≤ (ρ_A ⊗ σ_BC^{-1}).reindex(assoc^{-1}). This is derived from W_mat_sq_le_one by algebraic manipulation (conjugation and simplification).

```
theorem intermediate_ineq [Nonempty dA] [Nonempty dB] [Nonempty dC]
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (hρ : ρAB.mat.PosDef) (hσ : σBC.mat.PosDef) :
    ρAB ⊗ₖ (σBC.traceLeft)⁻¹ ≤
```

```
    (ρAB.traceRight ⊗ₖ σBC⁻¹).reindex (Equiv.prodAssoc dA dB dC).symm
```

theorem operator_ineq_SSA [source]

***Operator extension of SSA** (Main result of Lin-Kim-Hsieh). For positive definite ρ_AB and σ_BC: `ρ_A⁻¹ ⊗ σ_BC ≤ ρ_AB⁻¹ ⊗ σ_C` where ρ_A = Tr_B(ρ_AB) and σ_C = Tr_B(σ_BC), and the RHS is reindexed via the associator `(dA × dB) × dC ≃ dA × (dB × dC)`.*

```
theorem operator_ineq_SSA [Nonempty dA] [Nonempty dB] [Nonempty dC]
    (ρAB : HermitianMat (dA × dB) ℂ) (σBC : HermitianMat (dB × dC) ℂ)
    (hρ : ρAB.mat.PosDef) (hσ : σBC.mat.PosDef) :
    ρAB.traceRight⁻¹ ⊗ₖ σBC ≤
      (ρAB⁻¹ ⊗ₖ σBC.traceLeft).reindex (Equiv.prodAssoc dA dB dC)
```

lemma inner_kron_one_eq_inner_traceRight [source]

```
private lemma inner_kron_one_eq_inner_traceRight
    (A : HermitianMat d₁ ℂ) (M : HermitianMat (d₁ × d₂) ℂ) :
    ⟪A ⊗ₖ (1 : HermitianMat d₂ ℂ), M⟫ = ⟪A, M.traceRight⟫
```

lemma inner_one_kron_eq_inner_traceLeft [source]

```
private lemma inner_one_kron_eq_inner_traceLeft
    (B : HermitianMat d₂ ℂ) (M : HermitianMat (d₁ × d₂) ℂ) :
    ⟪(1 : HermitianMat d₁ ℂ) ⊗ₖ B, M⟫ = ⟪B, M.traceLeft⟫
```

lemma hermitianMat_log_inv_eq_neg [source]

```
private lemma hermitianMat_log_inv_eq_neg
    (A : HermitianMat d₁ ℂ) [A.NonSingular] : A⁻¹.log = -A.log
```

lemma PosDef_assoc'_traceRight [source]

```
private lemma PosDef_assoc'_traceRight
    (ρ : MState (d₁ × d₂ × d₃)) (hρ : ρ.M.mat.PosDef) :
    ρ.assoc'.traceRight.M.mat.PosDef
```

lemma wm_inner_lhs [source]

```
private lemma wm_inner_lhs [Nonempty d₁] [Nonempty d₂] [Nonempty d₃]
    (ρ : MState (d₁ × d₂ × d₃)) :
    ⟪(-ρ.assoc'.traceRight.M.traceRight.log) ⊗ₖ (1 : HermitianMat (d₂ × d₃) ℂ) +
     (1 : HermitianMat d₁ ℂ) ⊗ₖ ρ.traceLeft.M.log, ρ.M⟫ =
    S_𝓤 ρ.traceRight - S_𝓤 ρ.traceLeft
```

lemma wm_inner_rhs [source]

```
private lemma wm_inner_rhs [Nonempty d₁] [Nonempty d₂] [Nonempty d₃]
    (ρ : MState (d₁ × d₂ × d₃)) :
    ⟪((-ρ.assoc'.traceRight.M.log) ⊗ₖ (1 : HermitianMat d₃ ℂ) +
     (1 : HermitianMat (d₁ × d₂) ℂ) ⊗ₖ ρ.traceLeft.M.traceLeft.log).reindex
      (Equiv.prodAssoc d₁ d₂ d₃), ρ.M⟫ =
```

```
    S_vn ρ.assoc'.traceRight - S_vn ρ.traceLeft.traceLeft
```

lemma S_vn_wm_pd [source]

Weak monotonicity (form 2) for positive definite states: $S(\rho_{12}) + S(\rho_{23}) \geq S(\rho_1) + S(\rho_3)$. Proved by applying operator_ineq_SSA, taking log, then inner product with ρ.

```
private lemma S_vn_wm_pd [Nonempty d₁] [Nonempty d₂] [Nonempty d₃]
    (ρ : MState (d₁ × d₂ × d₃)) (hρ : ρ.M.mat.PosDef) :
    S_vn ρ.traceRight + S_vn ρ.traceLeft.traceLeft ≤
    S_vn ρ.assoc'.traceRight + S_vn ρ.traceLeft
```

lemma approx_by_pd [source]

```
private lemma MState.approx_by_pd
    (ρ : MState d₁) :
    ∃ (ρn : ℕ → MState d₁), (∀ n, (ρn n).M.mat.PosDef) ∧
      Filter.Tendsto ρn Filter.atTop (nhds ρ)
```

lemma traceLeft_continuous [source]

```
@[fun_prop]
private lemma MState.traceLeft_continuous :
    Continuous (MState.traceLeft : MState (d₁ × d₂) → MState d₂)
```

lemma traceRight_continuous [source]

```
@[fun_prop]
private lemma MState.traceRight_continuous :
    Continuous (MState.traceRight : MState (d₁ × d₂) → MState d₁)
```

lemma assoc'_continuous [source]

```
@[fun_prop]
private lemma MState.assoc'_continuous :
    Continuous (MState.assoc' : MState (d₁ × d₂ × d₃) → MState ((d₁ × d₂) × d₃))
```

lemma S_vn_wm [source]

Weak monotonicity, version with partial traces.

```
lemma S_vn_wm (ρ : MState (d₁ × d₂ × d₃)) :
    S_vn ρ.traceRight + S_vn ρ.traceLeft.traceLeft ≤
    S_vn ρ.assoc'.traceRight + S_vn ρ.traceLeft
```

def perm_A_BCR' [source]

Permutation to relabel $(A\times B\times C)\times R$ as $A\times(B\times C\times R)$.

```
private def perm_A_BCR' (dA dB dC : Type*) :
    (dA × dB × dC) × (dA × dB × dC) ≃ dA × (dB × dC × (dA × dB × dC)) where
```

def ρBCR [source]

The BCR state: trace out A from the purification of ρ_ABC.

```
private def ρBCR (ρ : MState (dA × dB × dC)) : MState (dB × dC × (dA × dB × dC))
```

lemma S_BC_of_BCR_eq [source]

```
private lemma S_BC_of_BCR_eq (ρ : MState (dA × dB × dC)) :
    S_vn (ρBCR ρ).assoc'.traceRight = S_vn ρ.traceLeft
```

def perm_AB_CR' [source]

Equivalence to relabel the purification as (dA × dB) × (dC × R).

```
private def perm_AB_CR' (dA dB dC : Type*) :
    (dA × dB × dC) × (dA × dB × dC) ≃ (dA × dB) × (dC × (dA × dB × dC)) where
```

lemma BCR_traceLeft_eq_purify_traceLeft [source]

```
private lemma BCR_traceLeft_eq_purify_traceLeft (ρ : MState (dA × dB × dC)) :
    (ρBCR ρ).traceLeft =
    ((MState.pure ρ.purify).relabel (perm_AB_CR' dA dB dC).symm).traceLeft
```

lemma purify_AB_traceRight_eq [source]

```
private lemma purify_AB_traceRight_eq (ρ : MState (dA × dB × dC)) :
    S_vn ((MState.pure ρ.purify).relabel (perm_AB_CR' dA dB dC).symm).traceRight =
    S_vn ρ.assoc'.traceRight
```

lemma S_CR_of_BCR_eq [source]

The CR-marginal of ρBCR has the same entropy as the AB-marginal of ρ.

```
private lemma S_CR_of_BCR_eq (ρ : MState (dA × dB × dC)) :
    S_vn (ρBCR ρ).traceLeft = S_vn ρ.assoc'.traceRight
```

lemma S_B_of_BCR_eq [source]

```
private lemma S_B_of_BCR_eq (ρ : MState (dA × dB × dC)) :
    S_vn (ρBCR ρ).traceRight = S_vn ρ.traceLeft.traceRight
```

lemma S_R_of_BCR_eq [source]

```
private lemma S_R_of_BCR_eq (ρ : MState (dA × dB × dC)) :
    S_vn (ρBCR ρ).traceLeft.traceLeft = S_vn ρ
```

theorem S_vn_strong_subadditivity [source]

Strong subadditivity on a tripartite system

```
theorem S_vn_strong_subadditivity (ρ123 : MState (d1 × d2 × d3)) :
    let ρ12
```

theorem S_{vn}_subadditivity [source]

"Ordinary" subadditivity of von Neumann entropy

```
theorem S_vn_subadditivity (ρ : MState (d₁ × d₂)) :
    S_vn ρ ≤ S_vn ρ.traceRight + S_vn ρ.traceLeft
```

theorem S_{vn}_pure_tripartite_triangle [source]

Triangle inequality for pure tripartite states: $S(A) \leq S(B) + S(C)$.

```
theorem S_vn_pure_tripartite_triangle (ψ : Ket ((d₁ × d₂) × d₃)) :
    S_vn (MState.pure ψ).traceRight.traceRight ≤
    S_vn (MState.pure ψ).traceRight.traceLeft + S_vn (MState.pure ψ).traceLeft
```

theorem S_{vn}_triangle_ineq_one_way [source]

One direction of the Araki-Lieb triangle inequality: $S(A) \leq S(B) + S(AB)$.

```
theorem S_vn_triangle_ineq_one_way (ρ : MState (d₁ × d₂)) : S_vn ρ.traceRight ≤ S_vn ρ.traceLeft +
    S_vn ρ
```

theorem S_{vn}_triangle_subaddivity [source]

Araki-Lieb triangle inequality on von Neumann entropy

```
theorem S_vn_triangle_subaddivity (ρ : MState (d₁ × d₂)) :
    abs (S_vn ρ.traceRight - S_vn ρ.traceLeft) ≤ S_vn ρ
```

theorem S_{vn}_weak_monotonicity [source]

Weak monotonicity of quantum conditional entropy: $S(A|B) + S(A|C) \geq 0$.

```
theorem S_vn_weak_monotonicity (ρ : MState (dA × dB × dC)) :
    let ρAB
```

theorem qConditionalEnt_strong_subadditivity [source]

Strong subadditivity, stated in terms of conditional entropies. Also called the data processing inequality. $H(A|BC) \leq H(A|B)$.

```
theorem qConditionalEnt_strong_subadditivity (ρ₁₂₃ : MState (d₁ × d₂ × d₃)) :
    qConditionalEnt ρ₁₂₃ ≤ qConditionalEnt ρ₁₂₃.assoc'.traceRight
```

theorem qMutualInfo_strong_subadditivity [source]

Strong subadditivity, stated in terms of quantum mutual information. $I(A;BC) \geq I(A;B)$.

```
theorem qMutualInfo_strong_subadditivity (ρ₁₂₃ : MState (d₁ × d₂ × d₃)) :
    qMutualInfo ρ₁₂₃ ≥ qMutualInfo ρ₁₂₃.assoc'.traceRight
```

theorem qcmi_nonneg [source]

The quantum conditional mutual information `QCMI` is nonnegative.

```
theorem qcmi_nonneg (ρ : MState (dA × dB × dC)) :
    0 ≤ qcmi ρ
```

theorem qcmi_le_2_log_dim [source]

The quantum conditional mutual information `QCMI ρABC` is at most 2 log dA.

```
theorem qcmi_le_2_log_dim (ρ : MState (dA × dB × dC)) :
    qcmi ρ ≤ 2 * Real.log (Fintype.card dA)
```

theorem qcmi_le_2_log_dim' [source]

The quantum conditional mutual information `QCMI ρABC` is at most 2 log dC.

```
theorem qcmi_le_2_log_dim' (ρ : MState (dA × dB × dC)) :
    qcmi ρ ≤ 2 * Real.log (Fintype.card dC)
```

2.42 QuantumInfo.Finite.Entropy.VonNeumann

QuantumInfo/Finite/Entropy/VonNeumann.lean on GitHub.

def S_{vn} [source]

Von Neumann entropy of a mixed state.

```
def S_vn (ρ : MState d) : ℝ
```

def qConditionalEnt [source]

The Quantum Conditional Entropy $S(\rho^A|\rho^B)$ is given by $S(\rho^{AB})$ - $S(\rho^B)$.

```
def qConditionalEnt (ρ : MState (dA × dB)) : ℝ
```

def qMutualInfo [source]

The Quantum Mutual Information I(A:B) is given by $S(\rho^A)$ + $S(\rho^B)$ - $S(\rho^{AB})$.

```
def qMutualInfo (ρ : MState (dA × dB)) : ℝ
```

def coherentInfo [source]

The Coherent Information of a state ρ passing through a channel Λ is the negative conditional entropy of the image under Λ of the purification of ρ.

```
def coherentInfo (ρ : MState d₁) (Λ : CPTPMap d₁ d₂) : ℝ
```

def qcmi [source]

The Quantum Conditional Mutual Information, I(A;C|B) = S(A|B) - S(A|BC).

```
def qcmi (ρ : MState (dA × dB × dC)) : ℝ
```

theorem S_{vn}_nonneg [source]

von Neumman entropy is nonnegative.

```
theorem S_vn_nonneg (ρ : MState d) : 0 ≤ S_vn ρ
```

theorem S_{vn}_le_log_d [source]

von Neumman entropy is at most log d.

```
theorem S_vn_le_log_d (ρ : MState d) : S_vn ρ ≤ Real.log (Finset.card Finset.univ (α := d))
```

theorem S_{vn}_of_pure_zero [source]

von Neumman entropy of pure states is zero.

```
@[simp]
theorem S_vn_of_pure_zero (ψ : Ket d) : S_vn (MState.pure ψ) = 0
```

theorem S_{vn}_eq_neg_trace_log [source]

```
theorem S_vn_eq_neg_trace_log (ρ : MState d) : S_vn ρ = -⟪ρ.M.log, ρ.M⟫
```

theorem S_{vn}_eq_trace_cfc_negMulLog [source]

Von Neumann entropy is the trace of the matrix function `x ↦ -x log x`.

```
theorem S_vn_eq_trace_cfc_negMulLog (ρ : MState d) :
    S_vn ρ = (ρ.M.cfc Real.negMulLog).trace
```

theorem S_{vn}_unit_zero [source]

```
@[simp]
theorem S_vn_unit_zero [Unique d] (ρ : MState d) : S_vn ρ = 0
```

theorem S_{vn}_relabel [source]

Von Neumann entropy is invariant under relabeling of the basis.

```
@[simp]
theorem S_vn_relabel (ρ : MState d₁) (e : d₂ ≃ d₁) :
    S_vn (ρ.relabel e) = S_vn ρ
```

theorem S_{vn}_of_SWAP_eq [source]

Von Neumann entropy is unchanged under SWAP. TODO: All unitaries

```
@[simp]
theorem S_vn_of_SWAP_eq (ρ : MState (d₁ × d₂)) : S_vn ρ.SWAP = S_vn ρ
```

theorem S_{vn}_of_assoc_eq [source]

Von Neumann entropy is unchanged under assoc.

```
@[simp]
theorem S_vn_of_assoc_eq (ρ : MState ((d₁ × d₂) × d₃)) : S_vn ρ.assoc = S_vn ρ
```

theorem S_vn_of_assoc'_eq [source]

Von Neumann entropy is unchanged under assoc'.

```
@[simp]
theorem S_vn_of_assoc'_eq (ρ : MState (d₁ × (d₂ × d₃))) : S_vn ρ.assoc' = S_vn ρ
```

theorem selfAdjointMap_Continuous [source]

```
@[fun_prop]
theorem selfAdjointMap_Continuous {𝕜 : Type*} [RCLike 𝕜] :
    Continuous (IsMaximalSelfAdjoint.selfadjMap : 𝕜 →+ ℝ)
```

theorem trace_Continuous [source]

```
@[fun_prop]
theorem HermitianMat.trace_Continuous {d 𝕜 : Type*} [Fintype d] [RCLike 𝕜]  :
    Continuous (HermitianMat.trace : HermitianMat d 𝕜 → ℝ)
```

theorem S_vn_continuous [source]

```
@[fun_prop]
theorem S_vn_continuous : Continuous (S_vn (d := d))
```

def vecToMat [source]

Convert a vector on a product space to a matrix.

```
private def vecToMat {d₁ d₂ : Type*} [Fintype d₁] [Fintype d₂] (v : d₁ × d₂ → ℂ) : Matrix d₁ d
    ₂ ℂ
```

lemma traceRight_eq_mul_conjTranspose [source]

The right partial trace of a pure state is the product of the matrix representation and its conjugate transpose.

```
private lemma traceRight_eq_mul_conjTranspose (ψ : Ket (d₁ × d₂)) :
    (MState.pure ψ).traceRight.M.val = (vecToMat ψ.vec) * (vecToMat ψ.vec).conjTranspose
```

lemma traceLeft_eq_transpose_mul_conj [source]

```
lemma traceLeft_eq_transpose_mul_conj (ψ : Ket (d₁ × d₂)) :
    (MState.pure ψ).traceLeft.M.val = (vecToMat ψ.vec).transpose * (vecToMat ψ.vec).map star
```

lemma traceLeft_eq_transpose_conjTranspose_mul [source]

*The left partial trace of `vecToMat` is the transpose of M^H * M.*

```
private lemma traceLeft_eq_transpose_conjTranspose_mul (ψ : Ket (d₁ × d₂)) :
    (MState.pure ψ).traceLeft.M.val =
    ((vecToMat ψ.vec).conjTranspose * (vecToMat ψ.vec)).transpose
```

lemma H_s_eq_of_nonzero_multiset_eq [source]

Shannon entropy is determined by the multiset of non-zero probabilities.

```
private lemma Hs_eq_of_nonzero_multiset_eq {α β : Type*} [Fintype α] [Fintype β]
  (d₁ : ProbDistribution α) (d₂ : ProbDistribution β)
  (h : (Finset.univ.val.map d₁.prob).filter (· ≠ 0) = (Finset.univ.val.map d₂.prob ).filter (·
    ≠ 0)) :
    Hs d₁ = Hs d₂
```

lemma nonzero_roots_eq_of_charpoly_eq_X_pow [source]

If two polynomials differ by a factor of X^k, their non-zero roots are the same.

```
private lemma nonzero_roots_eq_of_charpoly_eq_X_pow {R : Type*} [CommRing R] [IsDomain R]
  [DecidableEq R] (p q : Polynomial R) (k : ℕ) (h : p = Polynomial.X ^ k * q) :
    p.roots.filter (· ≠ 0) = q.roots.filter (· ≠ 0)
```

lemma nonzero_eigenvalues_eq_of_mul_comm [source]

The non-zero eigenvalues of AB and BA are the same.

```
private lemma nonzero_eigenvalues_eq_of_mul_comm {n m : Type*} [Fintype n] [Fintype m]
  [DecidableEq n] [DecidableEq m] (A : Matrix n m ℂ) (B : Matrix m n ℂ) :
    (A * B).charpoly.roots.filter (· ≠ 0) = (B * A).charpoly.roots.filter (· ≠ 0)
```

lemma S_{vn}_eq_of_nonzero_eigenvalues_eq [source]

If two states have the same non-zero eigenvalues (spectrum probabilities), they have the same von Neumann entropy.

```
private lemma Svn_eq_of_nonzero_eigenvalues_eq (ρ₁ : MState d₁) (ρ₂ : MState d₂)
    (h : (Finset.univ.val.map ρ₁.spectrum.prob).filter (· ≠ 0) =
      (Finset.univ.val.map ρ₂.spectrum.prob).filter (· ≠ 0)) :
    Svn ρ₁ = Svn ρ₂
```

lemma multiset_filter_map_ofReal_eq [source]

Filtering non-zero elements commutes with mapping `RCLike.ofReal` for multisets.

```
private lemma multiset_filter_map_ofReal_eq {R : Type*} [RCLike R] (M : Multiset ℝ) :
    (M.map (RCLike.ofReal : ℝ → R)).filter (· ≠ 0) = (M.filter (· ≠ 0)).map RCLike.ofReal
```

lemma charpoly_roots_filter_ne_zero_eq_eigenvalues_filter_ne_zero [source]

```
private lemma charpoly_roots_filter_ne_zero_eq_eigenvalues_filter_ne_zero {d : Type*} [Fintype
    d] [DecidableEq d] (A : Matrix d d ℂ) (hA : A.IsHermitian) :
    A.charpoly.roots.filter (· ≠ 0) = ((Finset.univ.val.map hA.eigenvalues).filter (· ≠ 0)).map
     RCLike.ofReal
```

lemma multiset_map_ofReal_injective [source]

Mapping `RCLike.ofReal` over a multiset is injective.

```
private lemma multiset_map_ofReal_injective {R : Type*} [RCLike R] {M N : Multiset ℝ} :
    M.map (RCLike.ofReal : ℝ → R) = N.map RCLike.ofReal ↔ M = N
```

lemma S_{vn}_eq_of_charpoly_roots_eq [source]

If the non-zero roots of the characteristic polynomials of two states are equal, their von Neumann entropies are equal.

```
private lemma S_vn_eq_of_charpoly_roots_eq (ρ₁ : MState d₁) (ρ₂ : MState d₂)
    (h : (ρ₁.M.1.charpoly.roots.filter (· ≠ 0)) = (ρ₂.M.1.charpoly.roots.filter (· ≠ 0))) :
    S_vn ρ₁ = S_vn ρ₂
```

theorem S_{vn}_of_partial_eq [source]

von Neumman entropies of the left- and right- partial trace of pure states are equal.

```
theorem S_vn_of_partial_eq (ψ : Ket (d₁ × d₂)) :
    S_vn (MState.pure ψ).traceLeft = S_vn (MState.pure ψ).traceRight
```

theorem qConditionalEnt_of_pure_symm [source]

Quantum conditional entropy is symmetric for pure states.

```
@[simp]
theorem qConditionalEnt_of_pure_symm (ψ : Ket (d₁ × d₂)) :
    qConditionalEnt (MState.pure ψ).SWAP = qConditionalEnt (MState.pure ψ)
```

theorem qMutualInfo_symm [source]

Quantum mutual information is symmetric.

```
@[simp]
theorem qMutualInfo_symm (ρ : MState (d₁ × d₂)) :
    qMutualInfo ρ.SWAP = qMutualInfo ρ
```

theorem S_{vn}_pure_complement [source]

For a pure state, the entropy of one subsystem equals the entropy of its complement, even after relabeling.

```
@[simp]
theorem S_vn_pure_complement (ψ : Ket d₁) (e : d₂ × d₃ ≃ d₁) :
    S_vn ((MState.pure ψ).relabel e).traceLeft = S_vn ((MState.pure ψ).relabel e).traceRight
```

2.43 QuantumInfo.Finite.MState

`QuantumInfo/Finite/MState.lean` on GitHub.

structure MState [source]

*A **mixed quantum state** is a PSD matrix with trace 1.*

We don't `extend (M : HermitianMat d ℂ)` because that gives an annoying thing where `M` is actually a `Subtype`, which means `ρ.M.foo` notation doesn't work.

```
@[ext]
structure MState (d : Type*) [Fintype d] [DecidableEq d] where
  M : HermitianMat d ℂ
  nonneg : 0 ≤ M
  tr : M.trace = 1
```

instance instCoe [source]

```
instance instCoe : Coe (MState d) (HermitianMat d ℂ)
```

def m [source]

The underlying `Matrix` in an MState. Prefer `MState.M` for the `HermitianMat`.

```
def m (ρ : MState d) : Matrix d d ℂ
```

theorem mat_M [source]

```
@[simp]
theorem mat_M : ρ.M.mat = ρ.m
```

theorem pos [source]

```
theorem pos (ρ : MState d) : 0 < ρ.M
```

def evalMStateM [source]

Positivity extension for `MState.M`: it is always positive (`0 < ρ.M`). Note: we must not call `whnfR` on `e` because `MState.M` is a structure projection (reducible), so `whnfR` would reduce it and destroy the pattern.

```
@[positivity MState.M _]
def evalMStateM : PositivityExt where eval {_u _α} _zα _pα e
```

theorem psd [source]

```
theorem psd : ρ.m.PosSemidef
```

theorem Hermitian [source]

Every mixed state is Hermitian.

```
theorem Hermitian : ρ.m.IsHermitian
```

theorem tr' [source]

```
@[simp]
theorem tr' : ρ.m.trace = 1
```

theorem ext_m [source]

```
theorem ext_m {ρ₁ ρ₂ : MState d} (h : ρ₁.m = ρ₂.m) : ρ₁ = ρ₂
```

theorem m_inj [source]

The map from mixed states to their matrices is injective

```
theorem m_inj : (MState.m (d := d)).Injective
```

theorem M_Injective [source]

```
theorem M_Injective : Function.Injective (MState.M (d := d))
```

theorem convex [source]

The matrices corresponding to MStates are `Convex ℝ`

```
theorem convex : Convex ℝ (Set.range (MState.M (d := d)))
```

instance instMixable [source]

```
instance instMixable : Mixable (HermitianMat d ℂ) (MState d) where
```

instance nonempty [source]

```
instance nonempty : Nonempty d
```

theorem eigenvalue_nonneg [source]

```
theorem eigenvalue_nonneg : ∀ i, 0 ≤ ρ.Hermitian.eigenvalues i
```

theorem eigenvalue_le_one [source]

```
theorem eigenvalue_le_one : ∀ i, ρ.Hermitian.eigenvalues i ≤ 1
```

theorem le_one [source]

```
theorem le_one : ρ.M ≤ 1
```

instance Inner Prob (MState d) [source]

The inner product of two MState's, as a real number between 0 and 1.

```
scoped instance : Inner Prob (MState d) where
```

theorem inner_def [source]

```
theorem inner_def : ⟪ρ, σ⟫_Prob = (⟪ρ.M, σ.M⟫,
    inner_ge_zero ρ.nonneg σ.nonneg,
    (inner_le_mul_trace ρ.nonneg σ.nonneg).trans (by simp))
```

theorem val_inner [source]

```
theorem val_inner : (⟪ρ, σ⟫_Prob : ℝ) = ⟪ρ.M, σ.M⟫
```

def exp_val_ℂ [source]

```
def exp_val_ℂ (T : Matrix d d ℂ) : ℂ
```

def exp_val [source]

*The **expectation value** of an operator on a quantum state.*

```
def exp_val (T : HermitianMat d ℂ) : ℝ
```

theorem exp_val_nonneg [source]

```
theorem exp_val_nonneg {T : HermitianMat d ℂ} (h : 0 ≤ T) : 0 ≤ ρ.exp_val T
```

theorem exp_val_zero [source]

```
@[simp]
theorem exp_val_zero : ρ.exp_val 0 = 0
```

theorem exp_val_one [source]

```
@[simp]
theorem exp_val_one : ρ.exp_val 1 = 1
```

theorem exp_val_le_one [source]

```
theorem exp_val_le_one {T : HermitianMat d ℂ} (h : T ≤ 1) : ρ.exp_val T ≤ 1
```

theorem exp_val_prob [source]

```
theorem exp_val_prob {T : HermitianMat d ℂ} (h : 0 ≤ T ∧ T ≤ 1) :
    0 ≤ ρ.exp_val T ∧ ρ.exp_val T ≤ 1
```

theorem exp_val_sub [source]

```
theorem exp_val_sub (A B : HermitianMat d ℂ) :
    ρ.exp_val (A - B) = ρ.exp_val A - ρ.exp_val B
```

theorem exp_val_eq_zero_iff [source]

If a PSD observable `A` has expectation value of 0 on a state `ρ`, it must entirely contain the support of `ρ` in its kernel.

```
theorem exp_val_eq_zero_iff {A : HermitianMat d ℂ} (hA₁ : 0 ≤ A) :
    ρ.exp_val A = 0 ↔ ρ.M.support ≤ A.ker
```

theorem exp_val_eq_one_iff [source]

If an observable `A` has expectation value of 1 on a state `ρ`, it must entirely contain the support of `ρ` in its 1-eigenspace.

```
theorem exp_val_eq_one_iff {A : HermitianMat d ℂ} (hA₂ : A ≤ 1) :
    ρ.exp_val A = 1 ↔ ρ.M.support ≤ (1 - A).ker
```

theorem exp_val_add [source]

```
theorem exp_val_add (A B : HermitianMat d ℂ) :
    ρ.exp_val (A + B) = ρ.exp_val A + ρ.exp_val B
```

theorem exp_val_smul [source]

```
@[simp]
theorem exp_val_smul (r : ℝ) (A : HermitianMat d ℂ) :
    ρ.exp_val (r • A) = r * ρ.exp_val A
```

theorem exp_val_le_exp_val [source]

```
@[gcongr]
theorem exp_val_le_exp_val (ρ : MState d) {A B : HermitianMat d ℂ} (h : A ≤ B) :
    ρ.exp_val A ≤ ρ.exp_val B
```

def pure [source]

A mixed state can be constructed as a pure state arising from a ket.

```
def pure (ψ : Ket d) : MState d where
```

theorem pure_apply [source]

```
@[simp]
theorem pure_apply {i j : d} : (pure ψ).m i j = (ψ i) * conj (ψ j)
```

theorem pure_mul_self [source]

```
theorem pure_mul_self : (pure ψ).m * (pure ψ).m = (pure ψ : Matrix d d ℂ)
```

def purity [source]

The purity of a state is Tr[ρ^2]. This is a `Prob`, because it is always between zero and one.

```
def purity (ρ : MState d) : Prob
```

def spectrum [source]

The eigenvalue spectrum of a mixed quantum state, as a `Distribution`.

```
def spectrum (ρ : MState d) : ProbDistribution d
```

theorem spectrum_pure_eq_constant [source]

The spectrum of a pure state is (1,0,0,...), i.e. a constant distribution.

```
theorem spectrum_pure_eq_constant :
    ∃ i, (pure ψ).spectrum = ProbDistribution.constant i
```

theorem pure_of_constant_spectrum [source]

If the spectrum of a mixed state is (1,0,0...) i.e. a constant distribution, it is a pure state.

```
theorem pure_of_constant_spectrum (h : ∃ i, ρ.spectrum = ProbDistribution.constant i) :
    ∃ ψ, ρ = pure ψ
```

theorem pure_iff_constant_spectrum [source]

A state ρ is pure iff its spectrum is (1,0,0,...) i.e. a constant distribution.

```
theorem pure_iff_constant_spectrum : (∃ ψ, ρ = pure ψ) ↔
    ∃ i, ρ.spectrum = ProbDistribution.constant i
```

theorem pure_iff_purity_one [source]

```
theorem pure_iff_purity_one : (∃ ψ, ρ = pure ψ) ↔ ρ.purity = 1
```

theorem spectralDecomposition [source]

```
theorem spectralDecomposition (ρ : MState d) :
    ∃ (ψs : d → Ket d), ρ.M = ∑ i, (ρ.spectrum i : ℝ) • (MState.pure (ψs i)).M
```

def prod [source]

```
def prod (ρ₁ : MState d₁) (ρ₂ : MState d₂) : MState (d₁ × d₂) where
```

theorem prod_inner_prod [source]

```
theorem prod_inner_prod (ξ1 ψ1 : MState d₁) (ξ2 ψ2 : MState d₂) :
    ⟪ξ1 ⊗ᴹ ξ2, ψ1 ⊗ᴹ ψ2⟫_Prob = ⟪ξ1, ψ1⟫_Prob * ⟪ξ2, ψ2⟫_Prob
```

theorem pure_prod_pure [source]

The product of pure states is a pure product state , `Ket.prod`.

```
theorem pure_prod_pure (ψ₁ : Ket d₁) (ψ₂ : Ket d₂) : pure (ψ₁ ⊗^ϕ ψ₂) = (pure ψ₁) ⊗ᴹ (pure ψ₂)
```

def ofClassical [source]

A representation of a classical distribution as a quantum state, diagonal in the given basis.

```
def ofClassical (dist : ProbDistribution d) : MState d where
```

theorem coe_ofClassical [source]

```
@[simp]
theorem coe_ofClassical (dist : ProbDistribution d) :
    (ofClassical dist).M = diagonal ℂ (dist ·)
```

theorem ofClassical_pow [source]

```
theorem ofClassical_pow (dist : ProbDistribution d) (p : ℝ) :
    (ofClassical dist).M ^ p = diagonal ℂ (fun i ↦ (dist i) ^ p)
```

def uniform [source]

The maximally mixed state.

```
def uniform [Nonempty d] : MState d
```

instance instUnique [source]

There is exactly one state on a dimension-1 system.

```
instance instUnique [Unique d] : Unique (MState d) where
```

instance instInhabited [source]

There exists a mixed state for every nonempty `d`. Here, the maximally mixed one is chosen.

```
instance instInhabited [Nonempty d] : Inhabited (MState d) where
```

theorem M_default [source]

```
@[simp]
theorem M_default [Unique d] : (default : MState d).M = 1
```

def traceLeft [source]

Partial tracing out the left half of a system.

```
@[simps]
def traceLeft (ρ : MState (d₁ × d₂)) : MState d₂ where
```

def traceRight [source]

Partial tracing out the right half of a system.

```
@[simps]
def traceRight (ρ : MState (d₁ × d₂)) : MState d₁ where
```

theorem traceLeft_prod_eq [source]

Taking the direct product on the left and tracing it back out gives the same state.

```
@[simp]
theorem traceLeft_prod_eq (ρ₁ : MState d₁) (ρ₂ : MState d₂) : (ρ₁ ⊗ᴹ ρ₂).traceLeft = ρ₂
```

theorem traceRight_prod_eq [source]

Taking the direct product on the right and tracing it back out gives the same state.

```
@[simp]
theorem traceRight_prod_eq (ρ₁ : MState d₁) (ρ₂ : MState d₂) : (ρ₁ ⊗ᴹ ρ₂).traceRight = ρ₁
```

theorem spectrum_prod [source]

Spectrum of direct product. There is a permutation σ so that the spectrum of the direct product of ρ_1 and ρ_2, as permuted under σ, is the pairwise products of the spectra of ρ_1 and ρ_2.

```
theorem spectrum_prod (ρ₁ : MState d₁) (ρ₂ : MState d₂) : ∃(σ : d₁ × d₂ ≃ d₁ × d₂),
    ∀i, ∀j, (ρ₁ ⊗ᴹ ρ₂).spectrum (σ (i, j)) = (ρ₁.spectrum i) * (ρ₂.spectrum j)
```

theorem sInf_spectrum_prod [source]

```
theorem sInf_spectrum_prod (ρ : MState d) (σ : MState d₂) :
    sInf (_root_.spectrum ℝ (ρ ⊗ᴹ σ).m) = sInf (_root_.spectrum ℝ ρ.m) * sInf (_root_.spectrum
    ℝ σ.m)
```

def IsSeparable [source]

A mixed state is separable iff it can be written as a convex combination of product mixed states.

```
def IsSeparable (ρ : MState (d₁ × d₂)) : Prop
```

theorem IsSeparable_prod [source]

A product state `MState.prod` is separable.

```
theorem IsSeparable_prod (ρ₁ : MState d₁) (ρ₂ : MState d₂) : IsSeparable (ρ₁ ⊗ᴹ ρ₂)
```

theorem eq_of_sum_eq_pure [source]

```
theorem eq_of_sum_eq_pure {d : Type*} [Fintype d] [DecidableEq d]
    {ι : Type*} {s : Finset ι} {p : ι → ℝ} {ρs : ι → MState d}
    {ρ : MState d} (h_pure : ρ.purity = 1) (h_sum : ρ.M = ∑ i ∈ s, p i • (ρs i).M)
```

```
    (hp_nonneg : ∀ i ∈ s, 0 ≤ p i) (hp_sum : ∑ i ∈ s, p i = 1) (i : ι) (hi : i ∈ s) (hpi : 0 <
    p i) :
    ρs i = ρ
```

theorem purity_prod [source]

```
theorem purity_prod {d₁ d₂ : Type*} [Fintype d₁] [Fintype d₂] [DecidableEq d₁] [DecidableEq d₂]
    (ρ₁ : MState d₁) (ρ₂ : MState d₂) : (ρ₁ ⊗ᴹ ρ₂).purity = ρ₁.purity * ρ₂.purity
```

theorem pure_eq_pure_iff [source]

```
theorem pure_eq_pure_iff {d : Type*} [Fintype d] [DecidableEq d] (ψ φ : Ket d) :
    pure ψ = pure φ ↔ ∃ z : ℂ, ‖z‖ = 1 ∧ ψ.vec = z • φ.vec
```

theorem PhaseEquiv_iff_pure_eq [source]

Two kets are phase-equivalent if and only if their pure states are equal.

```
theorem PhaseEquiv_iff_pure_eq {d : Type*} [Fintype d] [DecidableEq d] (ψ φ : Ket d) :
    Ket.PhaseEquiv.r ψ φ ↔ MState.pure ψ = MState.pure φ
```

def pureQ [source]

`MState.pure` descends to the quotient `KetUpToPhase`.

```
def pureQ {d : Type*} [Fintype d] [DecidableEq d] : KetUpToPhase d → MState d
```

theorem pureQ_mk [source]

```
@[simp]
theorem pureQ_mk {d : Type*} [Fintype d] [DecidableEq d] (ψ : Ket d) :
    pureQ (Quotient.mk Ket.PhaseEquiv ψ) = MState.pure ψ
```

theorem pureQ_injective [source]

```
theorem pureQ_injective {d : Type*} [Fintype d] [DecidableEq d] : Function.Injective (pureQ (d
    := d))
```

theorem pure_separable_imp_IsProd [source]

```
theorem pure_separable_imp_IsProd {d₁ d₂ : Type*} [Fintype d₁] [Fintype d₂] [DecidableEq d₁] [
    DecidableEq d₂]
    (ψ : Ket (d₁ × d₂)) (h : IsSeparable (pure ψ)) : ψ.IsProd
```

theorem pure_separable_iff_IsProd [source]

A pure state is separable iff the ket is a product state.

```
theorem pure_separable_iff_IsProd (ψ : Ket (d₁ × d₂)) :
    IsSeparable (pure ψ) ↔ ψ.IsProd
```

theorem pure_iff_rank_eq_one [source]

A mixed state is pure if and only if its rank is 1.

```
theorem pure_iff_rank_eq_one {d : Type*} [Fintype d] [DecidableEq d] (ρ : MState d) :
    (∃ ψ, ρ = pure ψ) ↔ ρ.m.rank = 1
```

theorem IsProd_iff_rank_eq_one [source]

A ket on a product space is a product state if and only if its coefficient matrix has rank 1.

```
theorem Ket.IsProd_iff_rank_eq_one {d₁ d₂ : Type*} [Fintype d₁] [Fintype d₂] [DecidableEq d₁] [
    DecidableEq d₂]
    (ψ : Ket (d₁ × d₂)) :
    ψ.IsProd ↔ (Matrix.of (fun i j => ψ (i, j))).rank = 1
```

theorem pure_separable_iff_traceLeft_pure [source]

A pure state is separable iff the partial trace on the left is pure.

```
theorem pure_separable_iff_traceLeft_pure (ψ : Ket (d₁ × d₂)) : IsSeparable (pure ψ) ↔
    ∃ ψ₁, pure ψ₁ = (pure ψ).traceLeft
```

def purify [source]

The purification of a mixed state. Always uses the full dimension of the Hilbert space (d) to purify, so e.g. an existing pure state with d=4 still becomes d=16 in the purification. The defining property is `MState.traceRight_of_purify`; see also `MState.purify'` for the bundled version.

```
def purify (ρ : MState d) : Ket (d × d) where
```

theorem purify_spec [source]

The defining property of purification, that tracing out the purifying system gives the original mixed state.

```
@[simp]
theorem purify_spec (ρ : MState d) : (pure ρ.purify).traceRight = ρ
```

def purifyX [source]

`MState.purify` bundled with its defining property `MState.traceRight_of_purify`.

```
def purifyX (ρ : MState d) : { ψ : Ket (d × d) // (pure ψ).traceRight = ρ }
```

def MState.relabel [source]

```
@[simps]
def relabel (ρ : MState d₁) (e : d₂ ≃ d₁) : MState d₂ where
```

theorem relabel_m [source]

```
@[simp]
theorem relabel_m (ρ : MState d₁) (e : d₂ ≃ d₁) :
    (ρ.relabel e).m = ρ.m.submatrix e e
```

theorem relabel_refl [source]

```
@[simp]
theorem relabel_refl {d : Type*} [Fintype d] [DecidableEq d] (ρ : MState d) :
    ρ.relabel (Equiv.refl d) = ρ
```

theorem relabel_pure_exists [source]

Relabeling a pure state by a bijection yields another pure state.

```
theorem relabel_pure_exists (ψ : Ket d₁) (e : d₂ ≃ d₁) :
    ∃ ψ' : Ket d₂, (pure ψ).relabel e = pure ψ'
```

theorem relabel_relabel [source]

```
@[simp]
theorem relabel_relabel {d d₂ d₃ : Type*}
    [Fintype d] [DecidableEq d] [Fintype d₂] [DecidableEq d₂] [Fintype d₃] [DecidableEq d₃]
    (ρ : MState d) (e : d₂ ≃ d) (e₂ : d₃ ≃ d₂) : (ρ.relabel e).relabel e₂ = ρ.relabel (e₂.trans
     e)
```

theorem eq_relabel_iff [source]

```
theorem eq_relabel_iff {d₁ d₂ : Type u} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq
     d₂]
    (ρ : MState d₁) (σ : MState d₂) (h : d₁ ≃ d₂) :
    ρ = σ.relabel h ↔ ρ.relabel h.symm = σ
```

theorem relabel_comp [source]

```
theorem relabel_comp {d₁ d₂ d₃ : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq
     d₂]
      [Fintype d₃] [DecidableEq d₃] (ρ : MState d₁) (e : d₂ ≃ d₁) (f : d₃ ≃ d₂) :
    (ρ.relabel e).relabel f = ρ.relabel (f.trans e)
```

theorem relabel_cast [source]

```
theorem relabel_cast {d₁ d₂ : Type u} [Fintype d₁] [DecidableEq d₁]
    [Fintype d₂] [DecidableEq d₂]
       (ρ : MState d₁) (e : d₂ = d₁) :
    ρ.relabel (Equiv.cast e) = cast (by have := e.symm; congr <;> (apply Subsingleton.helim;
    congr)) ρ
```

theorem spectrum_relabel [source]

```
@[simp]
theorem spectrum_relabel {ρ : MState d} (e : d₂ ≃ d) :
```

```
    _root_.spectrum ℝ (ρ.relabel e).m = _root_.spectrum ℝ ρ.m
```

theorem purity_relabel [source]

The purity of a state is invariant under relabeling of the basis.

```
@[simp]
theorem purity_relabel (ρ : MState d₁) (e : d₂ ≃ d₁) : (ρ.relabel e).purity = ρ.purity
```

def SWAP [source]

The heterogeneous SWAP gate that exchanges the left and right halves of a quantum system. This can apply even when the two "halves" are of different types, as opposed to (say) the SWAP gate on quantum circuits that leaves the qubit dimensions unchanged. Notably, it is not unitary.

```
def SWAP (ρ : MState (d₁ × d₂)) : MState (d₂ × d₁)
```

lemma multiset_spectrum_relabel_eq [source]

The multiset of values in the spectrum of a relabeled state is the same as the multiset of values in the spectrum of the original state.

```
lemma multiset_spectrum_relabel_eq {d₁ d₂ : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [
    DecidableEq d₂]
    (ρ : MState d₁) (e : d₂ ≃ d₁) :
    Multiset.map (ρ.relabel e).spectrum Finset.univ.val = Multiset.map ρ.spectrum Finset.univ.
    val
```

def spectrum_SWAP [source]

```
def spectrum_SWAP (ρ : MState (d₁ × d₂)) : ∃ e, ρ.SWAP.spectrum.relabel e = ρ.spectrum
```

theorem SWAP_SWAP [source]

```
@[simp]
theorem SWAP_SWAP (ρ : MState (d₁ × d₂)) : ρ.SWAP.SWAP = ρ
```

theorem traceLeft_SWAP [source]

```
@[simp]
theorem traceLeft_SWAP (ρ : MState (d₁ × d₂)) : ρ.SWAP.traceLeft = ρ.traceRight
```

theorem traceRight_SWAP [source]

```
@[simp]
theorem traceRight_SWAP (ρ : MState (d₁ × d₂)) : ρ.SWAP.traceRight = ρ.traceLeft
```

def assoc [source]

The associator that re-clusters the parts of a quantum system.

```
def assoc (ρ : MState ((d₁ × d₂) × d₃)) : MState (d₁ × d₂ × d₃)
```

def assoc' [source]

The associator that re-clusters the parts of a quantum system.

```
def assoc' (ρ : MState (d₁ × d₂ × d₃)) : MState ((d₁ × d₂) × d₃)
```

theorem assoc_assoc' [source]

```
@[simp]
theorem assoc_assoc' (ρ : MState (d₁ × d₂ × d₃)) : ρ.assoc'.assoc = ρ
```

theorem assoc'_assoc [source]

```
@[simp]
theorem assoc'_assoc (ρ : MState ((d₁ × d₂) × d₃)) : ρ.assoc.assoc' = ρ
```

theorem traceLeft_right_assoc [source]

```
@[simp]
theorem traceLeft_right_assoc (ρ : MState ((d₁ × d₂) × d₃)) :
    ρ.assoc.traceLeft.traceRight = ρ.traceRight.traceLeft
```

theorem traceRight_left_assoc' [source]

```
@[simp]
theorem traceRight_left_assoc' (ρ : MState (d₁ × d₂ × d₃)) :
    ρ.assoc'.traceRight.traceLeft = ρ.traceLeft.traceRight
```

theorem traceRight_assoc [source]

```
@[simp]
theorem traceRight_assoc (ρ : MState ((d₁ × d₂) × d₃)) :
    ρ.assoc.traceRight = ρ.traceRight.traceRight
```

theorem traceLeft_assoc' [source]

```
@[simp]
theorem traceLeft_assoc' (ρ : MState (d₁ × d₂ × d₃)) :
    ρ.assoc'.traceLeft = ρ.traceLeft.traceLeft
```

theorem traceLeft_left_assoc [source]

```
@[simp]
theorem traceLeft_left_assoc (ρ : MState ((d₁ × d₂) × d₃)) :
    ρ.assoc.traceLeft.traceLeft = ρ.traceLeft
```

theorem traceRight_right_assoc' [source]

```
@[simp]
theorem traceRight_right_assoc' (ρ : MState (d₁ × d₂ × d₃)) :
    ρ.assoc'.traceRight.traceRight = ρ.traceRight
```

theorem traceNorm_eq_1 [source]

```
@[simp]
theorem traceNorm_eq_1 (ρ : MState d) : ρ.m.traceNorm = 1
```

theorem relabel_kron [source]

```
theorem relabel_kron (ρ : MState d₁) (σ : MState d₂) (e : d₃ ≃ d₁) :
    ((ρ.relabel e) ⊗ᴹ σ) = (ρ ⊗ᴹ σ).relabel (e.prodCongr (Equiv.refl d₂))
```

theorem kron_relabel [source]

```
theorem kron_relabel (ρ : MState d₁) (σ : MState d₂) (e : d₃ ≃ d₂) :
    (ρ ⊗ᴹ σ.relabel e) = (ρ ⊗ᴹ σ).relabel ((Equiv.refl d₁).prodCongr e)
```

theorem prod_assoc [source]

```
theorem prod_assoc (ρ : MState d₁) (σ : MState d₂) (τ : MState d₃) :
    (ρ ⊗ᴹ (σ ⊗ᴹ τ)) = (ρ ⊗ᴹ σ ⊗ᴹ τ).relabel (Equiv.prodAssoc d₁ d₂ d₃).symm
```

instance TopologicalSpace (MState d) [source]

Mixed states inherit the subspace topology from matrices

```
instance : TopologicalSpace (MState d)
```

theorem toMat_IsEmbedding [source]

The projection from mixed states to their Hermitian matrices is an embedding

```
theorem toMat_IsEmbedding : Topology.IsEmbedding (MState.M (d := d)) where
```

instance T3Space (MState d) [source]

```
instance : T3Space (MState d)
```

instance CompactSpace (MState d) [source]

```
instance : CompactSpace (MState d)
```

instance MetricSpace (MState d) [source]

```
noncomputable instance : MetricSpace (MState d)
```

theorem dist_eq [source]

```
theorem dist_eq (x y : MState d) : dist x y = dist x.M y.M
```

instance BoundedSpace (MState d) [source]

```
instance : BoundedSpace (MState d) where
```

theorem Continuous_HermitianMat [source]

```
@[fun_prop]
theorem Continuous_HermitianMat : Continuous (MState.M (d := d))
```

theorem Continuous_Matrix [source]

```
@[fun_prop]
theorem Continuous_Matrix : Continuous (MState.m (d := d))
```

theorem image_M_isBounded [source]

```
theorem image_M_isBounded (S : Set (MState d)) : Bornology.IsBounded (MState.M '' S)
```

def piProd [source]

```
def piProd (ρi : (i:ι) → MState (dI i)) : MState ((i:ι) → dI i) where
```

def npow [source]

The n-copy "power" of a mixed state, with the standard basis indexed by pi types.

```
def npow (ρ : MState d) (n : ℕ) : MState (Fin n → d)
```

theorem kron [source]

```
theorem PosDef.kron {d₁ d₂ : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq d₂]
    {σ₁ : MState d₁} {σ₂ : MState d₂} (hσ₁ : σ₁.m.PosDef) (hσ₂ : σ₂.m.PosDef) : (σ₁ ⊗ᴹ σ₂).m.
    PosDef
```

theorem PosDef.relabel [source]

```
theorem PosDef.relabel {d₁ d₂ : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq
    d₂]
    {ρ : MState d₁} (hρ : ρ.m.PosDef) (e : d₂ ≃ d₁) : (ρ.relabel e).m.PosDef
```

theorem PosDef_mix [source]

If both states positive definite, so is their mixture.

```
theorem PosDef_mix {d : Type*} [Fintype d] [DecidableEq d] {σ₁ σ₂ : MState d}
    (hσ₁ : σ₁.m.PosDef) (hσ₂ : σ₂.m.PosDef) (p : Prob) : (p [σ₁ ↔ σ₂]).m.PosDef
```

theorem PosDef_mix_of_ne_zero [source]

If one state is positive definite and the mixture is nondegenerate, their mixture is also positive definite.

```
theorem PosDef_mix_of_ne_zero {d : Type*} [Fintype d] [DecidableEq d] {σ₁ σ₂ : MState d}
    (hσ₁ : σ₁.m.PosDef) (p : Prob) (hp : p ≠ 0) : (p [σ₁ ↔ σ₂]).m.PosDef
```

theorem PosDef_mix_of_ne_one [source]

If the second state is positive definite and the mixture is nondegenerate, their mixture is also positive definite.

```
theorem PosDef_mix_of_ne_one {d : Type*} [Fintype d] [DecidableEq d] {σ₁ σ₂ : MState d}
    (hσ₂ : σ₂.m.PosDef) (p : Prob) (hp : p ≠ 1) : (p [σ₁ ↔ σ₂]).m.PosDef
```

theorem uniform_posDef [source]

```
theorem uniform_posDef {d : Type*} [Nonempty d] [Fintype d] [DecidableEq d] :
    (uniform (d := d)).m.PosDef
```

theorem posDef_of_unique [source]

```
theorem posDef_of_unique {d : Type*} [Fintype d] [DecidableEq d] (ρ : MState d) [Unique d] : ρ.
    m.PosDef
```

2.44 QuantumInfo.Finite.Pinching

QuantumInfo/Finite/Pinching.lean on GitHub.

def pinching_kraus [source]

```
def pinching_kraus (ρ : MState d) : spectrum ℝ ρ.m → HermitianMat d ℂ
```

theorem pinching_kraus_commutes [source]

```
theorem pinching_kraus_commutes (ρ : MState d) (i : spectrum ℝ ρ.m) :
    Commute (pinching_kraus ρ i).mat ρ.m
```

theorem pinching_kraus_mul_self [source]

```
theorem pinching_kraus_mul_self (ρ : MState d) (i : spectrum ℝ ρ.m) :
    (pinching_kraus ρ i).mat * ρ.m = i.val • pinching_kraus ρ i
```

instance finite_spectrum_inst [source]

```
instance finite_spectrum_inst (ρ : MState d) : Fintype (spectrum ℝ ρ.m)
```

theorem pinching_kraus_orthogonal [source]

```
theorem pinching_kraus_orthogonal (ρ : MState d) {i j : spectrum ℝ ρ.m} (h : i ≠ j) :
    (pinching_kraus ρ i).mat * (pinching_kraus ρ j).mat = 0
```

theorem pinching_sq_eq_self [source]

The Kraus operators of the pinching channelare projectors: they square to themselves.

```
@[simp]
theorem pinching_sq_eq_self (ρ : MState d) (k) : (pinching_kraus ρ k) ^ 2  = pinching_kraus ρ k
```

theorem pinching_kraus_ortho [source]

The Kraus operators of the pinching channel are orthogonal projectors.

```
theorem pinching_kraus_ortho (ρ : MState d) (i j : spectrum ℝ ρ.m) :
    (pinching_kraus ρ i).mat * (pinching_kraus ρ j).mat = if i = j then (pinching_kraus ρ i).
    mat else 0
```

theorem pinching_sum [source]

```
theorem pinching_sum (ρ : MState d) : ∑ k, pinching_kraus ρ k = 1
```

def pinching_map [source]

```
def pinching_map (ρ : MState d) : CPTPMap d d ℂ
```

theorem pinchingMap_apply_M [source]

```
theorem pinchingMap_apply_M (σ ρ : MState d) : (pinching_map σ ρ).M =
  ⟨_, (MatrixMap.of_kraus_isCompletelyPositive
    (HermitianMat.mat ∘ pinching_kraus σ)).IsPositive.IsHermitianPreserving ρ.M.H⟩
```

theorem pinching_eq_sum_conj [source]

```
theorem pinching_eq_sum_conj (σ ρ : MState d) : (pinching_map σ ρ).M =
    ∑ k, (pinching_kraus σ k).mat * ρ.M * (pinching_kraus σ k).mat
```

theorem pinching_commutes_kraus [source]

```
theorem pinching_commutes_kraus (σ ρ : MState d) (i : spectrum ℝ σ.m) :
    Commute (pinching_map σ ρ).m (pinching_kraus σ i).mat
```

theorem pinching_commutes [source]

```
theorem pinching_commutes (ρ σ : MState d) :
    Commute (pinching_map σ ρ).m σ.m
```

theorem pinching_self [source]

```
@[simp]
theorem pinching_self (ρ : MState d) : pinching_map ρ ρ = ρ
```

theorem pinching_bound [source]

Lemma 3.10 of Hayashi's book "Quantum Information Theory - Mathematical Foundations". Also, Lemma 5 in https://arxiv.org/pdf/quant-ph/0107004. – Used in (S60)

```
theorem pinching_bound (ρ σ : MState d) : ρ.M ≤ (↑(Fintype.card (spectrum ℝ σ.m)) : ℝ) • (
    pinching_map σ ρ).M
```

theorem ker_le_ker_pinching_of_PosDef [source]

```
theorem ker_le_ker_pinching_of_PosDef (ρ σ : MState d) (hpos : σ.m.PosDef) : σ.M.ker ≤ (
    pinching_map σ ρ).M.ker
```

theorem pinching_idempotent [source]

```
theorem pinching_idempotent (ρ σ : MState d) :
    (pinching_map σ) (pinching_map σ ρ) = (pinching_map σ ρ)
```

theorem inner_cfc_pinching [source]

```
theorem inner_cfc_pinching (ρ σ : MState d) (f : ℝ → ℝ) :
    ⟪ρ.M, (pinching_map σ ρ).M.cfc f⟫ = ⟪(pinching_map σ ρ).M, (pinching_map σ ρ).M.cfc f⟫
```

theorem inner_cfc_pinching_right [source]

```
theorem inner_cfc_pinching_right (ρ σ : MState d) (f : ℝ → ℝ) :
    ⟪(pinching_map σ ρ).M, σ.M.cfc f⟫ = ⟪ρ.M, σ.M.cfc f⟫
```

theorem pinching_map_eq_sum_conj_hermitian [source]

```
theorem pinching_map_eq_sum_conj_hermitian (σ ρ : MState d) :
    (pinching_map σ ρ).M = ∑ k, ρ.M.conj (pinching_kraus σ k).mat
```

theorem pinching_map_ker_le [source]

```
theorem pinching_map_ker_le (ρ σ : MState d) : (pinching_map σ ρ).M.ker ≤ ρ.M.ker
```

theorem pinching_kraus_ker_of_ne_zero [source]

```
theorem pinching_kraus_ker_of_ne_zero {d : Type*} [Fintype d] [DecidableEq d]
    (σ : MState d) (v : d → ℂ) (hv : σ.m.mulVec v = 0)
    (k : spectrum ℝ σ.m) (hk : k.val ≠ 0) :
    (pinching_kraus σ k).mat *ᵥ v = 0
```

theorem ker_le_ker_pinching_map_ker [source]

```
theorem ker_le_ker_pinching_map_ker (ρ σ : MState d) (h : σ.M.ker ≤ ρ.M.ker) :
    σ.M.ker ≤ (pinching_map σ ρ).M.ker
```

theorem pinching_pythagoras [source]

Exercise 2.8 of Hayashi's book "A group theoretic approach to Quantum Information".

```
theorem pinching_pythagoras (ρ σ : MState d) :
    𝐃(ρ‖σ) = 𝐃(ρ‖pinching_map σ ρ) + 𝐃(pinching_map σ ρ‖σ)
```

2.45 QuantumInfo.Finite.POVM

QuantumInfo/Finite/POVM.lean on GitHub.

structure POVM [source]

A POVM is a (finite) collection of PSD matrices on the same Hilbert space that sum to the identity. Here `X` indexes the matrices, and `d` is the space dimension.

Applied to an `MState` on that on that space with `POVM.measure`, this produces a distribution of outcomes indexed by the same type as the collection.

This measurement action can be composed with `MState.of_classical`, in which case it is equal to a CPTP map `measurement_map`.

```
structure POVM (X : Type*) (d : Type*) [Fintype X] [Fintype d] [DecidableEq d] where
  mats : X → HermitianMat d ℂ
  nonneg : ∀ x, 0 ≤ (mats x)
  normalized : ∑ x, mats x = 1
```

def measurementMap [source]

The act of measuring is a quantum channel, that maps a `d`-dimensional quantum state to an `d × X`-dimensional quantum-classical state.

```
def measurementMap (Λ : POVM X d) : CPTPMap d (d × X) where
```

theorem measurementMap_apply_matrix [source]

```
theorem measurementMap_apply_matrix (Λ : POVM X d) (m : Matrix d d ℂ) :
  Λ.measurementMap.map m =  ∑ x : X,
    ((((Λ.mats x) ^ (1/2:ℝ)).mat * m * ((Λ.mats x)^(1/2:ℝ)).mat) ⊗ₖ Matrix.single x x 1)
```

theorem measurementMap_apply_hermitianMat [source]

```
theorem measurementMap_apply_hermitianMat (Λ : POVM X d) (m : HermitianMat d ℂ) :
  Λ.measurementMap.toHPMap m = ∑ x : X,
```

def measure [source]

A POVM leads to a distribution of outcomes on any given mixed state ρ.

```
def measure (Λ : POVM X d) (ρ : MState d) : ProbDistribution X
```

theorem traceLeft_measurementMap_eq_measure [source]

The quantum-classical `POVM.measurement_map`, gives a marginal on the right equal to `POVM.measure`.

```
theorem traceLeft_measurementMap_eq_measure (Λ : POVM X d) (ρ : MState d) :
    (Λ.measurementMap ρ).traceLeft = MState.ofClassical (Λ.measure ρ)
```

def measureDiscard [source]

The action of measuring a state with the POVM Λ, discarding the resulting state, and keeping the mixed state recording the outcome. This resulting state is purely diagonal, as given in `POVM.measureDiscard_apply`.

```
noncomputable def measureDiscard (Λ : POVM X d) : CPTPMap d X
```

theorem measureDiscard_apply [source]

```
theorem measureDiscard_apply (Λ : POVM X d) (ρ : MState d) :
    Λ.measureDiscard ρ = MState.ofClassical (Λ.measure ρ)
```

def measureForget [source]

The action of measuring a state with the POVM Λ, forgetting the measurement outcome, and keeping the disturbed state.

```
noncomputable def measureForget (Λ : POVM X d) : CPTPMap d d
```

2.46 QuantumInfo.Finite.Qubit.Basic

`QuantumInfo/Finite/Qubit/Basic.lean` on GitHub.

abbrev Qubit [source]

```
abbrev Qubit
```

def Z [source]

The Pauli Z gate on a qubit.

```
def Z : U[Qubit]
```

def X [source]

The Pauli X gate on a qubit.

```
def X : U[Qubit]
```

def Y [source]

The Pauli Y gate on a qubit.

```
def Y : U[Qubit]
```

def H [source]

The H gate, a Hadamard gate, on a qubit.

```
noncomputable def H : U[Qubit]
```

def S [source]

The S gate, or Rz(π/2) rotation on a qubit.

```
def S : U[Qubit]
```

def T [source]

The T gate, or Rz(π/4) rotation on a qubit.

```
noncomputable def T : U[Qubit]
```

theorem Z_sq [source]

```
@[simp]
theorem Z_sq : Z * Z = 1
```

theorem X_sq [source]

```
@[simp]
theorem X_sq : X * X = 1
```

theorem Y_sq [source]

```
@[simp]
theorem Y_sq : Y * Y = 1
```

theorem H_sq [source]

```
@[simp]
theorem H_sq : H * H = 1
```

theorem S_sq [source]

```
@[simp]
theorem S_sq : S * S = Z
```

theorem T_sq [source]

```
@[simp]
theorem T_sq : T * T = S
```

theorem X_Y_anticomm [source]

The anticommutator `{X,Y}` is zero. Marked simp as to put Pauli products in a canonical Y-X-Z order.

```
@[simp]
theorem X_Y_anticomm : X * Y = -Y * X
```

theorem Y_Z_anticomm [source]

The anticommutator `{Y,Z}` is zero.

```
theorem Y_Z_anticomm : Z * Y = -Y * Z
```

theorem Z_X_anticomm [source]

The anticommutator `{Z,X}` is zero.

```
theorem Z_X_anticomm : Z * X = -X * Z
```

theorem H_mul_X_eq_Z_mul_H [source]

```
@[simp]
theorem H_mul_X_eq_Z_mul_H : H * X = Z * H
```

theorem H_mul_Z_eq_X_mul_H [source]

```
@[simp]
theorem H_mul_Z_eq_X_mul_H : H * Z = X * H
```

theorem S_Z_comm [source]

```
@[simp]
theorem S_Z_comm : Z * S = S * Z
```

theorem T_Z_comm [source]

```
@[simp]
theorem T_Z_comm : Z * T = T * Z
```

theorem S_T_comm [source]

```
@[simp]
theorem S_T_comm : S * T = T * S
```

def controllize [source]

Given a unitary `U` on some Hilbert space `k`, we have the controllized version that acts on `Fin 2 ⊗ k` where `U` is conditionally applied if the first qubit is `1`.

```
def controllize (g : 𝐔[k]) : 𝐔[Qubit × k]
```

def CNOT [source]

Controlled-NOT gate on two qubits. The first qubit is the control and the second qubit is the target.

```
def CNOT : U[Fin 2 × Fin 2]
```

lemma CNOT_matrix [source]

The matrix representation of CNOT is the standard 4×4 permutation matrix.

```
lemma CNOT_matrix :
    Matrix.reindex finProdFinEquiv finProdFinEquiv CNOT.val =
      ![![(1:ℂ), 0, 0, 0],
        ![0, 1, 0, 0],
        ![0, 0, 0, 1],
        ![0, 0, 1, 0]]
```

theorem controllize_apply_zero_zero [source]

```
@[simp]
theorem controllize_apply_zero_zero : C[g] (0, j₁) (0, j₂) = (1 : U[k]) j₁ j₂
```

theorem controllize_apply_zero_one [source]

```
@[simp]
theorem controllize_apply_zero_one : C[g] (0, j₁) (1, j₂) = 0
```

theorem controllize_apply_one_zero [source]

```
@[simp]
theorem controllize_apply_one_zero : C[g] (1, j₁) (0, j₂) = 0
```

theorem controllize_apply_one_one [source]

```
@[simp]
theorem controllize_apply_one_one : C[g] (1, j₁) (1, j₂) = g j₁ j₂
```

theorem controllize_mul [source]

```
@[simp]
theorem controllize_mul (g₁ g₂ : U[k]) : C[g₁] * C[g₂] = C[g₁ * g₂]
```

theorem controllize_one [source]

```
@[simp]
theorem controllize_one : C[(1 : U[k])] = 1
```

theorem controllize_mul_inv [source]

```
@[simp]
theorem controllize_mul_inv : C[g] * C[g⁻¹] = 1
```

theorem X_controllize_X [source]

```
@[simp]
theorem X_controllize_X : (X ⊗ᵤ 1) * C[g] * (X ⊗ᵤ 1) = (1 ⊗ᵤ g) * C[g⁻¹]
```

2.47 QuantumInfo.Finite.ResourceTheory.FreeState

QuantumInfo/Finite/ResourceTheory/FreeState.lean on GitHub.

class ResourcePretheory [source]

A `ResourcePretheory` is a family of Hilbert spaces closed under tensor products, with an instance of `Fintype` and `DecidableEq` for each. It forms a pre-structure then on which to discuss resource theories. For instance, to talk about "two-party scenarios", we could write `ResourcePretheory (ℕ × ℕ)`, with `H (a,b) := (Fin a) × (Fin b)`.

The `Semigroup ι` structure means we have a way to take products of our labels of Hilbert spaces in a way that is associative (with actual equality). The `prodEquiv` lets us reinterpret between a product-labelled Hilbert spaces, and an actual pair of Hilbert spaces.

```
class ResourcePretheory (ι : Type*) extends Semigroup ι where
  H : ι → Type*
  [FinH : ∀ i, Fintype (H i)]
  [DecEqH : ∀ i, DecidableEq (H i)]
  [NonemptyH : ∀ i, Nonempty (H i)]
  prodEquiv i j : H (i * j) ≃ (H i) × (H j)
  hAssoc i j k :
    ((prodEquiv (i * j) k).trans <|
      ((prodEquiv i j).prodCongr (Equiv.refl (H k))).trans <|
      (Equiv.prodAssoc _ _ _).trans <|
      ((Equiv.refl (H i)).prodCongr ((prodEquiv j k).symm)).trans
      (prodEquiv i (j * k)).symm
    )
     = Equiv.cast (congrArg H <| mul_assoc i j k)
```

def prodRelabel [source]

The `prod` operation of `ResourcePretheory` gives the natural product operation on `MStates` that puts us in a new Hilbert space of the category. Accessible by the notation `ρ₁ ⊗ᵣ ρ₂`.

```
noncomputable def prodRelabel (ρ₁ : MState (H i)) (ρ₂ : MState (H j)) : MState (H (i * j))
```

theorem prodRelabel_assoc [source]

```
theorem prodRelabel_assoc (ρ₁ : MState (H i)) (ρ₂ : MState (H j)) (ρ₃ : MState (H k)) :
    ρ₁ ⊗ᵣ ρ₂ ⊗ᵣ ρ₃ ≍ ρ₁ ⊗ᵣ (ρ₂ ⊗ᵣ ρ₃)
```

theorem prodRelabel_relabel_cast_prod [source]

A `MState.relabel` can be distributed across a `prodRelabel`, if you have proofs that the factors correspond correctly.

```
theorem prodRelabel_relabel_cast_prod
    (ρ₁ : MState (H i)) (ρ₂ : MState (H j))
    (h : H (k * l) = H (i * j)) (hik : k = i) (hlj : l = j) :
    (ρ₁ ⊗ᵣ ρ₂).relabel (Equiv.cast h) =
    (ρ₁.relabel (Equiv.cast (congrArg H hik))) ⊗ᵣ (ρ₂.relabel (Equiv.cast (congrArg H hlj)))
```

def prodCPTPMap [source]

The `prod` operation of `ResourcePretheory` gives the natural product operation on `CPTPMaps`. Accessible by the notation `M₁ ⊗ᶜᵖᵣ M₂`.

```
noncomputable def prodCPTPMap (M₁ : CPTPMap (H i) (H j)) (M₂ : CPTPMap (H k) (H l)) :
    CPTPMap (H (i * k)) (H (j * l))
```

theorem prod [source]

```
theorem PosDef.prod {ρ : MState (H i)} {σ : MState (H j)} (hρ : ρ.m.PosDef) (hσ : σ.m.PosDef)
    : (ρ ⊗ᵣ σ).m.PosDef
```

theorem qRelEntropy_prodRelabel [source]

```
@[simp]
theorem qRelEntropy_prodRelabel (ρ₁ ρ₂ : MState (H i)) (σ₁ σ₂ : MState (H j)):
    𝐃(ρ₁ ⊗ᵣ σ₁‖ρ₂ ⊗ᵣ σ₂) = 𝐃(ρ₁‖ρ₂) + 𝐃(σ₁‖σ₂)
```

theorem sandwichedRelRentropy_prodRelabel [source]

```
@[simp]
theorem sandwichedRelRentropy_prodRelabel {α : ℝ} (ρ₁ ρ₂ : MState (H i)) (σ₁ σ₂ : MState (H j))
    :
    D̃ α(ρ₁ ⊗ᵣ σ₁‖ρ₂ ⊗ᵣ σ₂) = D̃ α(ρ₁‖ρ₂) + D̃ α(σ₁‖σ₂)
```

class UnitalPretheory [source]

A ResourcePretheory is `Unital` if it has a Hilbert space of size 1, i.e. $\mathbb{C}$.

```
class UnitalPretheory (ι : Type*) extends ResourcePretheory ι, MulOneClass ι, Unique (H 1)
    where
  prod_default {i} (ρ : MState (H i)) :
    (toResourcePretheory.prodRelabel ρ (Inhabited.default : MState (H 1))) ≍ ρ
  default_prod {i} (ρ : MState (H i)) :
    (toResourcePretheory.prodRelabel (Inhabited.default : MState (H 1)) ρ) ≍ ρ
```

instance Monoid ι [source]

```
instance : Monoid ι where
```

def spacePow [source]

Powers of spaces.

We define it for `Nat` in a `UnitalPretheory`. In principal this could be done for any `ResourcePretheory` and be defined for `PNat` so that we don't have zeroth powers. In

anticipation that we might some day want that, and that we might do everything with a non-equality associator, we keep this as its own definition and keep our own names for rewriting theorems where possible.

```
noncomputable def spacePow (i : ι) (n : ℕ) : ι
```

theorem spacePow_zero [source]

```
@[simp]
theorem spacePow_zero (i : ι) : i ^ 0 = 1
```

theorem spacePow_one [source]

```
@[simp]
theorem spacePow_one (i : ι) : i ^ 1 = i
```

theorem spacePow_succ [source]

```
theorem spacePow_succ (i : ι) (n : ℕ) : i ^ (n + 1) = (i ^ n) * i
```

theorem spacePow_add [source]

```
theorem spacePow_add (m n : ℕ) :
    i ^ (m + n) = (i ^ m) * (i ^ n)
```

theorem spacePow_mul [source]

```
theorem spacePow_mul (m n : ℕ) :
    i ^ (m * n) = (i ^ m) ^ n
```

def statePow [source]

Powers of states, using the resource theory's notion of product. Accessible via the notation `ρ ⊗ᵣ^[n]`.

```
noncomputable def statePow (ρ : MState (H i)) (n : ℕ) : MState (H (i ^ n))
```

theorem statePow_zero [source]

```
@[simp]
theorem statePow_zero (ρ : MState (H i)) : ρ ⊗ᵣ^[0] = default
```

theorem statePow_one [source]

```
@[simp]
theorem statePow_one (ρ : MState (H i)) : ρ ⊗ᵣ^[1] ≍ ρ
```

theorem statePow_succ [source]

```
theorem statePow_succ (ρ : MState (H i)) (n : ℕ) : ρ ⊗ᵣ^[n + 1] = ρ ⊗ᵣ^[n] ⊗ᵣ ρ
```

theorem statePow_add [source]

```
theorem statePow_add (ρ : MState (H i)) (m n : ℕ) : ρ ⊗ᵣ^[m + n] ≍ ρ ⊗ᵣ^[m] ⊗ᵣ ρ ⊗ᵣ^[n]
```

theorem statePow_add_relabel [source]

```
theorem statePow_add_relabel (ρ : MState (H i)) (m n : ℕ) :
    ρ ⊗ᵣ^[m + n] = (ρ ⊗ᵣ^[m] ⊗ᵣ ρ ⊗ᵣ^[n]).relabel (Equiv.cast (by congr; exact pow_add i m n))
```

theorem statePow_mul [source]

```
theorem statePow_mul (ρ : MState (H i)) (m n : ℕ) : ρ ⊗ᵣ^[m * n] ≍ (ρ ⊗ᵣ^[m]) ⊗ᵣ^[n]
```

theorem statePow_mul_relabel [source]

```
theorem statePow_mul_relabel {i : ι} (ρ : MState (H i)) (m n : ℕ) :
   ρ ⊗ᵣ^[m * n] = (ρ ⊗ᵣ^[m]) ⊗ᵣ^[n].relabel (Equiv.cast (congrArg H (pow_mul i m n)))
```

theorem PosDef.npow [source]

```
theorem PosDef.npow {ρ : MState (H i)} (hρ : ρ.m.PosDef) (n : ℕ)
    : (ρ ⊗ᵣ^[n]).m.PosDef
```

theorem statePow_rw [source]

```
theorem statePow_rw {n m : ℕ} (h : n = m) (ρ : MState (H i)) :
    ρ ⊗ᵣ^[n] = (ρ ⊗ᵣ^[m]).relabel (Equiv.cast (by congr))
```

theorem qRelEntropy_statePow [source]

```
@[simp]
theorem qRelEntropy_statePow (ρ σ : MState (H i)) (n : ℕ) :
    𝐃(ρ ⊗ᵣ^[n] ‖ σ  ⊗ᵣ^[n]) = n * 𝐃(ρ‖σ)
```

theorem sInf_spectrum_rprod [source]

```
theorem sInf_spectrum_rprod {j : ι} (ρ : MState (H i)) (σ : MState (H j)) :
    sInf (spectrum ℝ (ρ ⊗ᵣ σ).m) = sInf (spectrum ℝ ρ.m) * sInf (spectrum ℝ σ.m)
```

lemma sInf_spectrum_spacePow [source]

```
lemma sInf_spectrum_spacePow (σ : MState (H i)) (n : ℕ) :
    sInf (spectrum ℝ (σ ⊗ᵣ^[n]).m) = sInf (spectrum ℝ σ.m) ^ n
```

theorem sandwichedRelRentropy_statePow [source]

```
@[simp]
theorem sandwichedRelRentropy_statePow {α : ℝ} (ρ σ : MState (H i)) (n : ℕ) :
    D̃ α(ρ ⊗ᵣ^[n] ‖ σ ⊗ᵣ^[n]) = n * D̃ α(ρ‖σ)
```

class FreeStateTheory [source]

*A `FreeStateTheory` is a collection of mixed states (`MStates`) in a `ResourcePretheory` that obeys some necessary axioms: * For each Hilbert space `H i`, the free states are a closed, convex set * For each Hilbert space `H i`, there is a free state of full rank * If `ρᵢ ∈ H i` and `ρⱼ ∈ H j` are free, then `ρᵢ ⊗ ρⱼ` is free in `H (prod i j)`.*

```
class FreeStateTheory (ι : Type*) extends ResourcePretheory ι where
  IsFree : Set (MState (toResourcePretheory.H i))
  free_closed : IsClosed (@IsFree i)
  free_convex : Convex ℝ (MState.M '' (@IsFree i))
  free_prod {ρ₁ : MState (H i)} {ρ₂ : MState (H j)} (h₁ : IsFree ρ₁) (h₂ : IsFree ρ₂) : IsFree
    (ρ₁ ⊗ᵣ ρ₂)
  free_fullRank (i : ι) : open ComplexOrder in ∃ (ρ : MState (H i)), ρ.m.PosDef ∧ IsFree ρ
```

instance Inhabited_IsFree [source]

```
noncomputable instance Inhabited_IsFree : Inhabited (IsFree (i := i))
```

theorem of_unique [source]

```
theorem IsFree.of_unique [Unique (H i)] (ρ : MState (H i)) : ρ ∈ IsFree
```

theorem IsCompact_IsFree [source]

The set of free states is compact because it's a closed subset of a compact space.

```
theorem IsCompact_IsFree : IsCompact (IsFree (i := i))
```

theorem mix [source]

```
theorem IsFree.mix {ι : Type*} [FreeStateTheory ι] {i : ι} {σ₁ σ₂ : MState (H i)}
    (hσ₁ : IsFree σ₁) (hσ₂ : IsFree σ₂) (p : Prob) : IsFree (p [σ₁ ↔ σ₂])
```

class UnitalFreeStateTheory [source]

```
class UnitalFreeStateTheory (ι : Type*) extends FreeStateTheory ι, UnitalPretheory ι
```

theorem IsFree.npow [source]

```
theorem _root_.FreeStateTheory.IsFree.npow {i : ι} {ρ : MState (H i)}
    (hρ : IsFree ρ) (n : ℕ) : IsFree (ρ ⊗ᵣ^[n])
```

theorem relabel_cast_isFree [source]

```
@[simp]
theorem relabel_cast_isFree {i j : ι} (ρ : MState (H i)) (h : j = i) {h' : H j = H i}:
    ρ.relabel (Equiv.cast h') ∈ IsFree ↔ ρ ∈ IsFree
```

lemma relativeEntResource_ne_top [source]

In a `FreeStateTheory`, we have free states of full rank, therefore the minimum relative entropy of any state `ρ` to a free state is finite.

```
@[aesop (rule_sets := [finiteness]) safe]
lemma relativeEntResource_ne_top (ρ : MState (H i)) : ⨅ σ ∈ IsFree, 𝐃(ρ‖σ) ≠ ⊤
```

def RelativeEntResource [source]

```
noncomputable def RelativeEntResource : MState (H i) → ℝ≥0
```

theorem exists_isFree_relativeEntResource [source]

```
theorem exists_isFree_relativeEntResource (ρ : MState (H i)) :
    ∃ σ ∈ IsFree, 𝐃(ρ‖σ) = R_r ρ
```

theorem Subadditive [source]

```
theorem RelativeEntResource.Subadditive (ρ : MState (H i)) : Subadditive fun n ↦ R_r (ρ ⊗_r^[n])
```

def RegularizedRelativeEntResource [source]

```
noncomputable def RegularizedRelativeEntResource (ρ : MState (H i)) : ℝ≥0
```

theorem tendsto [source]

Lemma 5

```
theorem RelativeEntResource.tendsto (ρ : MState (H i)) :
    Filter.atTop.Tendsto (fun n ↦ R_r (ρ ⊗_r^[n]) / n) (𝓝 (R_r∞ ρ))
```

theorem tendsto_ennreal [source]

Alternate version of Lemma 5 which states the convergence with the `ENNReal` expression for `RelativeEntResource`, as opposed its `untop`-ped `NNReal` value.

```
theorem RelativeEntResource.tendsto_ennreal (ρ : MState (H i)) :
    Filter.atTop.Tendsto (fun n ↦ (⨅ σ ∈ IsFree, 𝐃(ρ ⊗_r^[n]‖σ)) / ↑n) (𝓝 (R_r∞ ρ))
```

def GlobalRobustness [source]

```
noncomputable def GlobalRobustness {i : ι} : MState (H i) → ℝ≥0
```

def IsAsymptoticallyNongenerating [source]

A sequence of operations `f_n` is asymptotically nongenerating if `lim_{n→∞} RG(f_n(ρ_n)) = 0`, where RG is `GlobalRobustness` and `ρ_n` is any sequence of free states. Equivalently, we can take the `max` (over operations and states) on the left-hand side inside the limit.

```
def IsAsymptoticallyNongenerating (dI dO : ι) (f : (n : ℕ) → CPTPMap (H (dI⊗^H[n])) (H (dO⊗^H[n
    ]))) : Prop
```

2.48 `QuantumInfo.Finite.ResourceTheory.HypothesisTesting`

`QuantumInfo/Finite/ResourceTheory/HypothesisTesting.lean` on GitHub.

def OptimalHypothesisRate [source]

The optimal hypothesis testing rate, for a tolerance ε: given a state ρ and a set of states S, the optimum distinguishing rate that allows a probability ε of errors.

```
noncomputable def OptimalHypothesisRate (ρ : MState d) (ε : Prob) (S : Set (MState d)) : Prob
```

instance iInf_Inhabited [source]

The space of strategies `T` in `OptimalHypothesisRate` is inhabited, we always have some valid strategy.

```
instance iInf_Inhabited (ρ : MState d) (ε : Prob) :
    Inhabited { m // ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤ m ∧ m ≤ 1 }
```

instance Inhabited {m | ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤ m ∧ m ≤ 1} [source]

```
instance (ρ : MState d) (ε : Prob) : Inhabited {m | ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤ m ∧ m ≤ 1}
```

theorem iInf_IsCompact [source]

The space of strategies `T` in `OptimalHypothesisRate` is compact.

```
theorem iInf_IsCompact (ρ : MState d) (ε : Prob) : IsCompact { m | ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤
    m ∧ m ≤ 1 }
```

theorem iInf_IsConvex [source]

The space of strategies `T` in `OptimalHypothesisRate` is convex.

```
theorem iInf_IsConvex (ρ : MState d) (ε : Prob) : Convex ℝ { m | ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤ m
    ∧ m ≤ 1 }
```

theorem of_empty [source]

When `S` is empty, the optimal hypothesis testing rate is zero.

```
@[simp]
theorem of_empty {ρ : MState d} (ε : Prob) : β_ ε(ρ‖∅) = 0
```

theorem le_sup_exp_val [source]

```
theorem le_sup_exp_val {ρ : MState d} (ε : Prob) {S : Set (MState d)}
    (m : HermitianMat d ℂ) (hExp : ρ.exp_val (1 - m) ≤ ε) (hm : 0 ≤ m ∧ m ≤ 1) :
    β_ ε(ρ‖S) ≤ ⨅ σ ∈ S, ⟨_, σ.exp_val_prob hm⟩
```

theorem le_of_subset [source]

```
theorem le_of_subset (ρ : MState d) (ε : Prob) {S1 S2 : Set (MState d)} (h : S1 ⊆ S2) :
    β_ ε(ρ‖S1) ≤ β_ ε(ρ‖S2)
```

theorem of_singleton [source]

```
theorem of_singleton {ρ σ : MState d} {ε : Prob} :
    β_ ε(ρ‖{σ}) =
      ⨅ T : { m : HermitianMat d ℂ // ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤ m ∧ m ≤ 1},
        ⟨_, σ.exp_val_prob T.2.right⟩
```

theorem negLog_le_singleton [source]

```
theorem negLog_le_singleton (ρ : MState d) (ε : Prob) (S : Set (MState d))
    (σ : MState d) (h : σ ∈ S) : -log β_ ε(ρ‖S) ≤ -log β_ ε(ρ‖{σ})
```

theorem singleton_le_exp_val [source]

```
theorem singleton_le_exp_val {ρ σ : MState d} {ε : Prob} (m : HermitianMat d ℂ)
    (hExp : ρ.exp_val (1 - m) ≤ ε) (hm : 0 ≤ m ∧ m ≤ 1) :
  β_ ε(ρ‖{σ}) ≤ ⟨_, σ.exp_val_prob hm⟩
```

theorem exists_min' [source]

There exists an optimal T for the hypothesis testing, that is, it's a minimum and not just an infimum. This states we have `1 - ε ≤ ρ.exp_val T`, but we can always "worsen" T to make that bound tight, which is `exists_min`.

```
theorem exists_min' (ρ : MState d) (ε : Prob) (S : Set (MState d)):
    ∃ (T : { m : HermitianMat d ℂ // ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤ m ∧ m ≤ 1}),
      (⨅ σ ∈ S, ⟨_, σ.exp_val_prob T.prop.right⟩ = β_ ε(ρ‖S))
      ∧ 1 - ε ≤ ρ.exp_val T
```

theorem exists_min [source]

There exists an optimal T for the hypothesis testing, that is, it's a minimum and not just an infimum. This tightens the `T` from `exists_min'` to a `⟪ρ,T⟫ = 1 - ε` bound.

```
theorem exists_min (ρ : MState d) (ε : Prob) (S : Set (MState d)):
    ∃ (T : { m : HermitianMat d ℂ // ρ.exp_val (1 - m) ≤ ε ∧ 0 ≤ m ∧ m ≤ 1}),
      (⨅ σ ∈ S, ⟨_, σ.exp_val_prob T.prop.right⟩ = β_ ε(ρ‖S))
      ∧ ρ.exp_val T = 1 - ε
```

theorem pos_of_lt_one [source]

When the allowed Type I error `ε` is less than 1 (so, we have some limit on our errors), and the kernel of the state `ρ` contains the kernel of some element in `S`, then the optimal hypothesis rate is positive - there is some lower bound on the type II errors we'll see. In other words, under these conditions, we cannot completely avoid type II errors.

```
theorem pos_of_lt_one {ρ : MState d} (S : Set (MState d))
  (hρ : ∃ σ ∈ S, σ.M.ker ≤ ρ.M.ker)
  {ε : Prob} (hε : ε < 1) : 0 < β_ ε(ρ‖S)
```

theorem Lemma3 [source]

```
theorem Lemma3 {ρ : MState d} (ε : Prob) {S : Set (MState d)} (hS₁ : IsCompact S)
    (hS₂ : Convex ℝ (MState.M '' S)) : ⨅ σ ∈ S, β_ ε(ρ‖{σ}) = β_ ε(ρ‖S)
```

theorem ker_diagonal_prob_eq_bot [source]

```
theorem ker_diagonal_prob_eq_bot {q : Prob} (hq₁ : 0 < q) (hq₂ : q < 1) :
    HermitianMat.ker (.diagonal ℂ (ProbDistribution.coin q ·)) = ⊥
```

theorem optimalHypothesisRate_antitone [source]

Lemma S1

```
theorem optimalHypothesisRate_antitone (ρ σ : MState d) (ℰ : CPTPMap d d₂) (ε : Prob) :
    β_ ε(ρ‖{σ}) ≤ β_ ε(ℰ ρ‖{ℰ σ})
```

theorem Ref81Lem5 [source]

This is from [Strong converse exponents for a quantum channel discrimination problem and quantum-feedback-assisted communication](https://doi.org/10.1007/s00220-016-2645-4), Lemma 5.

It seems like this is actually true for all $0 < \alpha$ (with appropriate modifications at $\alpha = 1$), but we only need it for the case of $1 < \alpha$.

```
theorem Ref81Lem5 (ρ σ : MState d) (ε : Prob) (hε : ε < 1) (α : ℝ) (hα : 1 < α) :
    -log β_ ε(ρ‖{σ}) ≤ D̃_ α(ρ‖σ) + -log (1 - ε) *
      (.ofNNReal ⟨α, zero_le_one.trans hα.le⟩) / (.ofNNReal ⟨α - 1, sub_nonneg_of_le hα.le⟩)
```

theorem rate_pos_of_smul_pos [source]

```
theorem rate_pos_of_smul_pos {ε : Prob} {d : Type*} [Fintype d] [DecidableEq d] {ρ σ₁ σ₂ :
    MState d}
    (hσ₂ : 0 < β_ ε(ρ‖{σ₂})) {c : ℝ} (hc : 0 < c) (hσ : c • σ₂ ≤ σ₁.M) : 0 < β_ ε(ρ‖{σ₁})
```

theorem rate_Continuous_singleton [source]

```
@[fun_prop]
theorem rate_Continuous_singleton {ε : Prob} {d : Type*} [Fintype d] [DecidableEq d] (ρ :
    MState d) :
    Continuous fun σ ↦ β_ ε(ρ‖{σ})
```

theorem optimalHypothesisRate_unique [source]

On the 1D Hilbert space, the optimal hypothesis testing rate is simply 1 - ε, since there's nothing to learn. (More generally this would hold whenever ρ=σ.) -

```
theorem optimalHypothesisRate_unique {d : Type*} [Fintype d] [DecidableEq d]
    (ε : Prob) (ρ σ : MState d) [Unique d] : β_ ε(ρ‖{σ}) = 1 - ε
```

2.49 QuantumInfo.Finite.ResourceTheory.ResourceTheory

QuantumInfo/Finite/ResourceTheory/ResourceTheory.lean on GitHub.

class ResourceTheory [source]

A quantum resource theory extends a `FreeStateTheory` with a collection of free operations. It is required that any state we can always prepare for free must be free, and this is all then the resource theory is "minimal", but we can have a more restricted set of operations.

```
class ResourceTheory (ι : Type*) extends FreeStateTheory ι where
  freeOps i j : Set (CPTPMap (H i) (H j))
  nongenerating {i j : ι} {f} (h : f ∈ freeOps i j) : ∀ ρ, IsFree ρ → IsFree (f ρ)
  free_id i : CPTPMap.id ∈ freeOps i i
  free_comp {i j k} (Y : freeOps j k) (X : freeOps i j) : Y.1.compose X.1 ∈ freeOps i k
```

def maximal [source]

Given a `FreeStateTheory`, there is a maximal set of free operations compatible with the free states. That is the set of all operations that don't generate non-free states from free states. We call this the maximal resource theory.

```
def maximal [FreeStateTheory ι] : ResourceTheory ι where
```

def IsMaximal [source]

A resource theory `IsMaximal` if it includes all non-generating operations.

```
def IsMaximal (r : ResourceTheory ι) : Prop
```

structure IsTensorial [source]

A resource theory `IsTensorial` if it includes tensor products of operations, creating free states, and discarding. This implies that includes a unit object.

```
structure IsTensorial [UnitalPretheory ι] : Prop where
  prod :  ∀ {i j k l : ι} {f g}, f ∈ freeOps i k → g ∈ freeOps j l → (f ⊗^{cp}_r g) ∈ freeOps (prod
     i j) (prod k l)
  create : ∀ {i : ι} (ρ), IsFree ρ → CPTPMap.replacement ρ ∈ freeOps Unital.unit i
  destroy : ∀ (i : ι), CPTPMap.destroy ∈ freeOps i Unital.unit
```

theorem maximal_IsMaximal [source]

The theory `ResourceTheory.maximal` always `IsMaximal`.

```
theorem maximal_IsMaximal [FreeStateTheory ι] : IsMaximal (maximal (ι := ι))
```

2.50 QuantumInfo.Finite.ResourceTheory.SteinsLemma

QuantumInfo/Finite/ResourceTheory/SteinsLemma.lean on GitHub.

def Lemma6_σn [source]

The \{}tilde{σ}_n defined in Lemma 6, also in equation (S40) in Lemma 7.

```
noncomputable def Lemma6_σn (m : ℕ) (σf : MState (H i)) (σₘ : MState (H (i ^ m))) : (n : ℕ) →
    MState (H (i ^ n))
```

theorem Lemma6_σn_IsFree [source]

```
theorem Lemma6_σn_IsFree {σ₁ : MState (H i)} {σₘ : (m : ℕ) → MState (H (i ^ m))} (hσ₁_free :
    IsFree σ₁)
    (hσₘ : ∀ (m : ℕ), σₘ m ∈ IsFree) (m n : ℕ) : Lemma6_σn m σ₁ (σₘ m) n ∈ IsFree
```

theorem Lemma6 [source]

Lemma 6 from the paper. We __did__ end up doing the version that "works also in the case of $\varepsilon = 0$", which is nice.

```
private theorem Lemma6 {m : ℕ} (hm : 0 < m) (ρ σf : MState (H i)) (σₘ : MState (H (i ^ m)))
    [σf.M.NonSingular] {ε : Prob} (hε : ε < 1) :
  Filter.atTop.limsup (fun n ↦ -log β_ ε(ρ ⊗ᵣ^[n]‖{Lemma6_σn m σf σₘ n}) / n) ≤ 𝐃(ρ ⊗ᵣ^[m]‖σₘ)
    / m
```

theorem LemmaS2liminf [source]

```
theorem LemmaS2liminf {ε3 : Prob} {ε4 : ℝ≥0} (hε4 : 0 < ε4)
  {d : ℕ → Type*} [∀ n, Fintype (d n)] [∀ n, DecidableEq (d n)] (ρ : (n : ℕ) → MState (d n)) (
    σ : (n : ℕ) → MState (d n))
  {Rinf : ℝ≥0} (hRinf : Rinf ≥ Filter.atTop.liminf (fun (n : ℕ) ↦ -log β_ ε3(ρ n‖{σ n}) / n))
  :
  (Filter.atTop.liminf (fun (n : ℕ) ↦ ⟪{(ρ n).M ≥ₚ (Real.exp (n * (Rinf + ε4))) • (σ n).M}, (ρ
    n).M⟫) ≤ 1 - ε3)
```

theorem LemmaS2limsup [source]

```
theorem LemmaS2limsup {ε3 : Prob} {ε4 : ℝ≥0} (hε4 : 0 < ε4)
  {d : ℕ → Type*} [∀ n, Fintype (d n)] [∀ n, DecidableEq (d n)] (ρ : (n : ℕ) → MState (d n)) (
    σ : (n : ℕ) → MState (d n))
  {Rsup : ℝ≥0} (hRsup : Rsup ≥ Filter.atTop.limsup (fun (n : ℕ) ↦ -log β_ ε3(ρ n‖{σ n}) / n))
  :
  (Filter.atTop.limsup (fun (n : ℕ) ↦ ⟪{ρ n ≥ₚ (Real.exp (n * (Rsup + ε4))) • (σ n).M}, ρ n⟫)
    ≤ 1 - ε3)
```

theorem LemmaS3_helper [source]

```
private theorem LemmaS3_helper {ε : Prob} {d : ℕ → Type*} [∀ n, Fintype (d n)] [∀ n,
    DecidableEq (d n)]
  (ρ σ₁ σ₂ : (n : ℕ) → MState (d n))
  (f : ℕ → ℝ≥0) (hσ : ∀ (i : ℕ), Real.exp (-f i) • (σ₂ i).M ≤ (σ₁ i)) (n : ℕ) :
    -log β_ ε(ρ n‖{σ₁ n}) ≤ -log β_ ε(ρ n‖{σ₂ n}) + f n
```

theorem LemmaS3_inf [source]

Lemma S3 from the paper. What they denote as σ_n and σ_n', we denote as σ_1 and σ_2. The `exp(-o(n))` we express as a function `f : ℕ → ℝ`, together with the fact that `f` is little-o of `n` (i.e. that `f =o[.atTop] id`), and then writing `exp(-f)`. We also split LemmaS3 into two parts, the `lim inf` part and the `lim sup` part. The theorem as written is true for any `f`, but we can restrict to nonnegative `f` (so, `ℕ → ℝ≥0`) which is easier to work with and more natural in the subsequent proofs.

```
private theorem LemmaS3_inf {ε : Prob}
```

```
    {d : ℕ → Type*} [∀ n, Fintype (d n)] [∀ n, DecidableEq (d n)]
    (ρ σ₁ σ₂ : (n : ℕ) → MState (d n))
    (f : ℕ → ℝ≥0) (hf : (f · : ℕ → ℝ) =o[.atTop] (· : ℕ → ℝ))
    (hσ : ∀ i, Real.exp (-f i) • (σ₂ i).M ≤ σ₁ i)
    :
    Filter.atTop.liminf (fun (n : ℕ) ↦ -log β_ ε(ρ n‖{σ₁ n}) / n) ≤
      Filter.atTop.liminf (fun (n : ℕ) ↦ -log β_ ε(ρ n‖{σ₂ n}) / n)
```

theorem LemmaS3_sup [source]

```
private theorem LemmaS3_sup {ε : Prob}
    {d : ℕ → Type*} [∀ n, Fintype (d n)] [∀ n, DecidableEq (d n)]
    (ρ σ₁ σ₂ : (n : ℕ) → MState (d n))
    (f : ℕ → ℝ≥0) (hf : (f · : ℕ → ℝ) =o[.atTop] (· : ℕ → ℝ))
    (hσ : ∀ i, Real.exp (-f i) • (σ₂ i).M ≤ σ₁ i)
    :
    Filter.atTop.limsup (fun (n : ℕ) ↦ -log β_ ε(ρ n‖{σ₁ n}) / n) ≤
      Filter.atTop.limsup (fun (n : ℕ) ↦ -log β_ ε(ρ n‖{σ₂ n}) / n)
```

def R1 [source]

```
private noncomputable def R1 (ρ : MState (H i)) (ε : Prob) : ℝ≥0∞
```

def R2 [source]

```
private noncomputable def R2 (ρ : MState (H i)) : ((n : ℕ) → IsFree (i := i ^ n)) → ℝ≥0∞
```

lemma commute_aux [source]

```
private lemma commute_aux (n : ℕ) {x : ℝ}
  {E 𝓔 σ : HermitianMat d ℂ} (h𝓔σ : Commute 𝓔.mat σ.mat)
  (hE : E = 1 - {Real.exp (n * x) • σ ≤ₚ 𝓔})
    : Commute ((1 / n : ℝ) • E).mat (𝓔.log - σ.log).mat
```

lemma rexp_mul_smul_proj_lt_mul_sub_le_mul_sub [source]

```
private lemma rexp_mul_smul_proj_lt_mul_sub_le_mul_sub {n : ℕ} {x : ℝ}
  {E 𝓔 σ : HermitianMat d ℂ} (h𝓔σ : Commute 𝓔.mat σ.mat) (hx : 0 < x)
  (h𝓔 : 𝓔.mat.PosSemidef) (hσ : σ.mat.PosDef)
  (hE : E = 1 - {Real.exp (n * x) • σ ≤ₚ 𝓔})
    : ⟪𝓔, ((1 / n : ℝ) • E) * (𝓔.log - σ.log)⟫ ≤ ⟪𝓔, x • E⟫
```

lemma rexp_mul_smul_proj_lt_mul_sub_le_mul_sub' [source]

```
private lemma rexp_mul_smul_proj_lt_mul_sub_le_mul_sub' {n : ℕ} {x : ℝ} {y : ℝ}
  {E 𝓔 σ : HermitianMat d ℂ} (h𝓔σ : Commute 𝓔.mat σ.mat)
  (h𝓔 : 𝓔.mat.PosSemidef) (hσ : σ.mat.PosDef)
  (hE : E = {Real.exp (n * y) • σ ≤ₚ 𝓔} - {Real.exp (n * x) • σ ≤ₚ 𝓔})
    : (1 / n : ℝ) • E.mat * (𝓔.log.mat - σ.log.mat) ≤ x • E
```

lemma f_image_bound [source]

```
private lemma f_image_bound (mineig : ℝ) (n : ℕ) (h : 0 < mineig) (hn : 0 < n) :
  let c : ℕ → ℝ
```

lemma c'_bounded [source]

```
private lemma c'_bounded {mineig : ℝ} {ε2 : ℕ → ℝ≥0}
    (hε2 : ∀ (n : ℕ), ε2 n < 1) (o : ℝ) :
  let c : ℕ → ℝ
```

def σ₁ [source]

```
private def σ₁ : MState (H i)
```

def σ₁_mineig [source]

```
private def σ₁_mineig
```

def σ₁_c [source]

The sequence c_n given in (S44). In order to handle when c = 0, we've replaced the (Real.log 3) / n term with (Real.log 3) / (max n 1).

```
private def σ₁_c (n : ℕ) : ℝ
```

def f_map [source]

The function f_n(λ) in (S45).

```
private def f_map (n : ℕ) (lam : ℝ) : ℝ
```

def «σ̃» [source]

```
private def «σ̃» :
    (n : ℕ) → MState (H (i ^ n))
```

def «σ⋆» [source]

```
private def «σ⋆» : (n : ℕ) → MState (H (i ^ n))
```

def σ' [source]

```
private def σ' : (n : ℕ) → MState (H (i ^ n))
```

def σ''_unnormalized [source]

```
private def σ''_unnormalized : (n : ℕ) → HermitianMat (H (i ^ n)) ℂ
```

theorem σ₁_pos [source]

```
private theorem σ₁_pos : (σ₁ i).m.PosDef
```

theorem σ₁_isFree [source]

```
private theorem σ₁_isFree : IsFree (σ₁ i)
```

theorem mineig_pos [source]

```
private theorem mineig_pos : 0 < σ₁_mineig i
```

theorem mineig_le_one [source]

```
private theorem mineig_le_one : σ₁_mineig i ≤ 1
```

theorem σ₁_c_pos [source]

```
private theorem σ₁_c_pos (n) : 0 < σ₁_c i n
```

theorem σ₁_c_div_lim [source]

```
private theorem σ₁_c_div_lim : Filter.atTop.Tendsto (fun n ↦ (σ₁_c i n) / n) (𝓝 0)
```

lemma σ₁_c_identity [source]

```
private lemma σ₁_c_identity {n : ℕ} (hn : 0 < n) :
    Real.exp (-σ₁_c i n) * (1 / 3) * σ₁_mineig i ^ n =
    Real.exp (-n * (σ₁_c i n + σ₁_c i n / n))
```

theorem σ₁_c_littleO [source]

```
theorem σ₁_c_littleO : (fun n : ℕ ↦ σ₁_c i n + Real.log 3) =o[Filter.atTop] (fun x ↦ (x : ℝ))
```

theorem log_le_f [source]

(S46), part 1

```
private theorem log_le_f (n : ℕ) (lam : ℝ) : Real.log lam ≤ f_map i n lam
```

theorem f_le_log [source]

(S46), part 2

```
private theorem f_le_log (n : ℕ) (lam : ℝ) : f_map i n lam < Real.log lam + σ₁_c i n
```

theorem le_exp_f [source]

```
private theorem le_exp_f (n : ℕ) (x : ℝ) (hx : 0 < x) : x ≤ Real.exp (f_map i n x)
```

theorem exp_f_le [source]

```
private theorem exp_f_le (n : ℕ) (x : ℝ) (hx : 0 < x) : Real.exp (f_map i n x) < Real.exp (σ₁_c
    i n) * x
```

theorem «σ̃_isFree» [source]

```
private theorem «σ̃_isFree» : IsFree («σ̃» m σ n)
```

theorem «σ⋆_free» [source]

```
private theorem «σ⋆_free» : IsFree («σ⋆» ρ ε n)
```

theorem «σ⋆_max» [source]

```
private theorem «σ⋆_max» :
    IsMaxOn (fun σ ↦ β_ ε(ρ ⊗ᵣ^[n]‖{σ})) IsFree («σ⋆» ρ ε n)
```

theorem σ'_free [source]

```
private theorem σ'_free : IsFree (σ' ρ ε m σ n)
```

theorem σ'_posdef [source]

```
private theorem σ'_posdef : (σ' ρ ε m σ n).m.PosDef
```

theorem hσ'n_eq_sum_third [source]

```
private theorem hσ'n_eq_sum_third : (σ' ρ ε m σ n).M =
    (1 / 3 : ℝ) • («σ̃» m σ n) + (1 / 3 : ℝ) • («σ⋆» ρ ε n) + (1 / 3 : ℝ) • ((σ₁ i) ⊗ᵣ^[n])
```

theorem hσ₁_le_σ' [source]

```
private theorem hσ₁_le_σ' : (1 / 3 : ℝ) • ((σ₁ i) ⊗ᵣ^[n]).M ≤ (σ' ρ ε m σ n).M
```

theorem σ''_unnormalized_PosDef [source]

```
private theorem σ''_unnormalized_PosDef : Matrix.PosDef (σ''_unnormalized ρ ε m σ n).mat
```

theorem σ''_tr_bounds [source]

```
private theorem σ''_tr_bounds : 1 ≤ (σ''_unnormalized ρ ε m σ n).trace ∧
    (σ''_unnormalized ρ ε m σ n).trace < Real.exp (σ₁_c i n)
```

def σ'' [source]

```
private def σ'' : (n : ℕ) → MState (H (i ^ n))
```

lemma σ''_posdef [source]

```
private lemma σ''_posdef n : (σ'' ρ ε m σ n).M.mat.PosDef
```

lemma σ'_le_σ'' [source]

```
private lemma σ'_le_σ'' (n) : Real.exp (-σ₁_c i n) • (σ' ρ ε m σ n).M ≤ σ'' ρ ε m σ n
```

lemma σ''_le_σ' [source]

```
private lemma σ''_le_σ' (n) : σ'' ρ ε m σ n ≤ Real.exp (σ₁_c i n) • (σ' ρ ε m σ n).M
```

theorem «σ''_ge_σ⋆» [source]

```
private theorem «σ''_ge_σ⋆» n : σ'' ρ ε m σ n ≥ (Real.exp (-σ₁_c i n) / 3) • («σ⋆» ρ ε n).M
```

theorem «σ''_ge_σ̃» [source]

```
private theorem «σ''_ge_σ̃» n : σ'' ρ ε m σ n ≥ (Real.exp (-σ₁_c i n) / 3) • («σ̃» m σ n).M
```

theorem σ''_ge_σ₁ [source]

```
private theorem σ''_ge_σ₁ n : σ'' ρ ε m σ n ≥ (Real.exp (-σ₁_c i n) / 3) • ((σ₁ i) ⊗ᵣ^[n]).M
```

abbrev ε₀_func [source]

```
private abbrev ε₀_func (ε' : Prob) : ℝ
```

theorem EquationS88 [source]

```
private theorem EquationS88 (ρ : MState (H i)) (σ : (n : ℕ) → ↑IsFree) {ε ε' : Prob}
  (hR1R2 : R1 ρ ε < R2 ρ σ) (hR1 : R1 ρ ε ≠ ⊤) (hR2 : R2 ρ σ ≠ ⊤) (hε₀_1 : 0 < ε₀_func ρ ε σ
    ε') (m : ℕ)
  :
  let ℰ
```

theorem EquationS62 [source]

```
private theorem EquationS62
    (ρ : MState (H i)) (σ : (n : ℕ) → IsFree (i := i ^ n))
    {ε ε' : Prob} (hε'₁ : 0 < ε') (hε'₂ : ε' < ε) (hε : ε < 1)
    (hR1R2 : R1 ρ ε < R2 ρ σ) (hR1 : R1 ρ ε ≠ ⊤) (hR2 : R2 ρ σ ≠ ⊤)
    (hε₀ : 0 < ε₀_func ρ ε σ ε') (hε₀' : (R1 ρ ε).toReal ≤ (R2 ρ σ).toReal + ε₀_func ρ ε σ ε')
    (m : ℕ) (hm : m ≥ 1 ∧ 𝐃(ρ ⊗ᵣ^[m]‖↑(σ m)) / ↑m < R2 ρ σ + (.ofNNReal ⟨ε₀_func ρ ε σ ε', hε₀.
    le⟩))
    :
  let ℰ n
```

theorem Lemma7 [source]

Lemma 7 from the paper. We write `ε'` for their `\{}tilde{ε}`.

```
private theorem Lemma7 (ρ : MState (H i)) {ε : Prob} (hε : 0 < ε ∧ ε < 1) (σ : (n : ℕ) → IsFree
     (i := i ^ n)) :
    (R2 ρ σ ≥ R1 ρ ε) →
    ∀ ε' : Prob, (hε' : 0 < ε' ∧ ε' < ε) →
    ∃ σ' : (n : ℕ) → IsFree (i := i ^ n),
    R2 ρ σ' - R1 ρ ε ≤ .ofNNReal (1 - ε' : Prob) * (R2 ρ σ - R1 ρ ε)
```

def Lemma7_improver [source]

Lemma 7 gives us a way to repeatedly "improve" a sequence σ to one with a smaller gap between R2 and R1. The paper paints this as pretty much immediate from Lemma7, but we need to handle the case where R2 is below R1.

```
private noncomputable def Lemma7_improver (ρ : MState (H i)) {ε : Prob} (hε : 0 < ε ∧ ε < 1) {
    ε' : Prob} (hε' : 0 < ε' ∧ ε' < ε) :
```

theorem Lemma7_gap [source]

The Lemma7_improver does its job at shrinking the gap.

```
theorem Lemma7_gap (ρ : MState (H i)) {ε : Prob} (hε : 0 < ε ∧ ε < 1) {ε' : Prob} (hε' : 0 < ε'
     ∧ ε' < ε) :
    ∀ σ,
      letI σ'
```

theorem GeneralizedQSteinsLemma [source]

Theorem 1 in https://arxiv.org/pdf/2408.02722v3

```
theorem GeneralizedQSteinsLemma {i : ι} (ρ : MState (H i)) {ε : Prob} (hε : 0 < ε ∧ ε < 1) :
    Filter.atTop.Tendsto (fun n ↦ -log β_ ε(ρ ⊗ᵣ^[n]‖IsFree) / n) (𝒩 (R_r∞ ρ))
```

theorem limit_hypotesting_eq_limit_rel_entropy [source]

Theorem 4, which is also called the Generalized quantum Stein's lemma in Hayashi & Yamasaki. What they state as an equality of limits, which don't exist per se in Mathlib, we state as the existence of a number (which happens to be `RegularizedRelativeEntResource`) to which both sides converge.

```
theorem limit_hypotesting_eq_limit_rel_entropy (ρ : MState (H i)) (ε : Prob) (hε : 0 < ε ∧ ε <
    1) :
    ∃ d : ℝ≥0,
      Filter.atTop.Tendsto (fun n ↦ -log β_ ε(ρ ⊗ᵣ^[n]‖IsFree) / n) (𝒩 d)
      ∧
      Filter.atTop.Tendsto (fun n ↦ (⨅ σ ∈ IsFree, 𝐃(ρ ⊗ᵣ^[n]‖σ)) / n) (𝒩 d)
```

2.51 QuantumInfo.Finite.Unitary

QuantumInfo/Finite/Unitary.lean on GitHub.

theorem trace_conj_unitary [source]

```
@[simp]
theorem trace_conj_unitary : (conj U.val A).trace = A.trace
```

theorem le_conj_unitary [source]

```
@[simp]
theorem le_conj_unitary : A.conj U.val ≤ B.conj U ↔ A ≤ B
```

theorem inner_conj_unitary [source]

```
@[simp]
theorem inner_conj_unitary : ⟪A.conj U.val, B.conj U.val⟫ = ⟪A, B⟫
```

theorem eigenvalues_conj [source]

The eigenvalues of a Hermitian matrix conjugated by a unitary matrix are the same as the eigenvalues of the original matrix.

```
@[simp]
theorem eigenvalues_conj:(A.conj U.val).H.eigenvalues = A.H.eigenvalues
```

def U_conj [source]

Conjugate a state by a unitary matrix (applying the unitary as an evolution).

```
def U_conj (ρ : MState d) (U : 𝐔[d]) : MState d where
```

theorem U_conj_spectrum_eq [source]

You might think this should only be true up to permutation, so that it would read like `∃ σ : Equiv.Perm d, (ρ.U_conj U).spectrum = ρ.spectrum.relabel σ`. But since eigenvalues of a matrix are always canonically sorted, this is actually an equality.

```
@[simp]
theorem U_conj_spectrum_eq (ρ : MState d) (U : 𝐔[d]) :
    (ρ.U_conj U).spectrum = ρ.spectrum
```

theorem inner_U_conj [source]

```
@[simp]
theorem inner_U_conj (ρ σ : MState d) (U : 𝐔[d]) : ⟪U ◁ ρ, U ◁ σ⟫_Prob = ⟪ρ, σ⟫_Prob
```

theorem no_cloning [source]

*The **No-cloning theorem**, saying that if states `ψ` and `φ` can both be perfectly cloned using a unitary `U` and a fiducial state `f`, and they aren't identical (their inner product is less than 1), then the two states must be orthogonal to begin with. In short: only orthogonal states can be simultaneously cloned.*

```
theorem no_cloning {U : 𝐔[d × d]}
  (hψ : U ◁ pure (ψ ⊗^ϕ f) = pure (ψ ⊗^ϕ ψ))
  (hφ : U ◁ pure (φ ⊗^ϕ f) = pure (φ ⊗^ϕ φ))
  (H : ⟪pure ψ, pure φ⟫_Prob < (1 : ℝ)) :
    ⟪pure ψ, pure φ⟫_Prob = (0 : ℝ)
```

2.52 QuantumInfo.ForMathlib.ContinuousLinearMap

QuantumInfo/ForMathlib/ContinuousLinearMap.lean on GitHub.

theorem range_zero [source]

```
@[simp]
theorem range_zero [RingHomSurjective σ] : (0 : M →SL[σ] M₂).range = ⊥
```

theorem ker_zero [source]

```
@[simp]
theorem ker_zero : (0 : M →SL[σ] M₂).ker = ⊤
```

theorem ker_mk [source]

```
theorem ker_mk (f : M →ₛₗ[σ] M₂) (hf : Continuous f.toFun) :
    (ContinuousLinearMap.mk f hf).ker = LinearMap.ker f
```

theorem support_eq_sup_eigenspace_nonzero [source]

The support of a Hermitian matrix is the sum of its nonzero eigenspaces.

```
theorem support_eq_sup_eigenspace_nonzero (A : EuclideanSpace 𝕜 n →L[𝕜] EuclideanSpace 𝕜 n)
    (hA : A.IsSymmetric) : A.range = ⨆ μ ≠ 0, Module.End.eigenspace A μ
```

2.53 QuantumInfo.ForMathlib.ContinuousSup

QuantumInfo/ForMathlib/ContinuousSup.lean on GitHub.

theorem sSup_image_eq_sSup_image_closure [source]

```
theorem sSup_image_eq_sSup_image_closure {f : β → α}
  (hS : IsCompact (closure S)) (hf : Continuous f) :
    sSup (f '' S) = sSup (f '' closure S)
```

theorem sInf_image_eq_sInf_image_closure [source]

```
theorem sInf_image_eq_sInf_image_closure {f : β → α} (hS : IsCompact (closure S)) (hf :
    Continuous f) :
    sInf (f '' S) = sInf (f '' closure S)
```

theorem closure_continuous_sSup [source]

A version of `IsCompact.continuous_sSup` with a slightly weaker hypothesis on the set `K`, that its closure is compact (but the set itself need not be).

```
theorem closure_continuous_sSup (hS : IsCompact (closure S)) (hf : Continuous ↑f) :
    Continuous fun x ↦ sSup (f x '' S)
```

theorem closure_continuous_sInf [source]

A version of `IsCompact.continuous_sInf` with a slightly weaker hypothesis on the set `K`, that its closure is compact (but the set itself need not be).

```
theorem closure_continuous_sInf (hS : IsCompact (closure S)) (hf : Continuous ↑f) :
    Continuous fun x ↦ sInf (f x '' S)
```

theorem IsCompact.continuous_iSup [source]

```
theorem continuous_iSup (hS : IsCompact (closure S)) (hf : Continuous ↿f) :
    Continuous fun x ↦ ⨆ y : S, f x y
```

theorem IsCompact.continuous_iInf [source]

```
theorem continuous_iInf (hS : IsCompact (closure S)) (hf : Continuous ↿f) :
    Continuous fun x ↦ ⨅ y : S, f x y
```

theorem continuous_sSup [source]

Similar to `IsCompact.continuous_sSup`, but taking a bounded set in the bornology instead of a compact set.

```
theorem continuous_sSup (hS : Bornology.IsBounded S) (hf : Continuous ↿f) :
    Continuous fun x ↦ sSup (f x '' S)
```

theorem continuous_sInf [source]

Similar to `IsCompact.continuous_sInf`, but taking a bounded set in the bornology instead of a compact set.

```
theorem continuous_sInf (hS : Bornology.IsBounded S) (hf : Continuous ↿f) :
    Continuous fun x ↦ sInf (f x '' S)
```

theorem IsBounded.continuous_iSup [source]

```
theorem continuous_iSup (hS : Bornology.IsBounded S) (hf : Continuous ↿f) :
    Continuous fun x ↦ ⨆ y : S, f x y
```

theorem IsBounded.continuous_iInf [source]

```
theorem continuous_iInf (hS : Bornology.IsBounded S) (hf : Continuous ↿f) :
    Continuous fun x ↦ ⨅ y : S, f x y
```

theorem LinearMap.continuous_iSup [source]

For bilinear maps in suitably well-behaved spaces with `IsModuleTopology`, taking the supremum in one argument is still `Continuous`, by `Bornology.IsBounded.continuous_iSup`.

```
theorem continuous_iSup {E F 𝕜 : Type*}
  [CommRing 𝕜] [TopologicalSpace 𝕜] [IsTopologicalRing 𝕜] [ConditionallyCompleteLinearOrder 𝕜]
     [OrderTopology 𝕜]
  [AddCommGroup E] [TopologicalSpace E] [Module 𝕜 E] [IsModuleTopology 𝕜 E]
  [AddCommGroup F] [Module 𝕜 F] [PseudoMetricSpace F] [ProperSpace F] [Module.Finite 𝕜 F] [
    IsModuleTopology 𝕜 F]
  (f : E →ₗ[𝕜] F →ₗ[𝕜] 𝕜) {S : Set F} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨆ y : S, f x y
```

theorem LinearMap.continuous_iInf [source]

For bilinear maps in suitably well-behaved spaces with `IsModuleTopology`, taking the infimum in one argument is still `Continuous`, by `Bornology.IsBounded.continuous_iInf`.

```
theorem continuous_iInf {E F 𝕜 : Type*}
  [CommRing 𝕜] [TopologicalSpace 𝕜] [IsTopologicalRing 𝕜] [ConditionallyCompleteLinearOrder 𝕜]
     [OrderTopology 𝕜]
  [AddCommGroup E] [TopologicalSpace E] [Module 𝕜 E] [IsModuleTopology 𝕜 E]
  [AddCommGroup F] [Module 𝕜 F] [PseudoMetricSpace F] [ProperSpace F] [Module.Finite 𝕜 F] [
    IsModuleTopology 𝕜 F]
  (f : E →ₗ[𝕜] F →ₗ[𝕜] 𝕜) {S : Set F} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨅ y : S, f x y
```

theorem continuous_iSup' [source]

A specialization of `LinearMap.continuous_iSup` to finite dimensional spaces, in place of requiring a (non-instance) `IsModuleTopology`.

```
theorem continuous_iSup' {E F 𝕜 : Type*}
  [NontriviallyNormedField 𝕜] [ConditionallyCompleteLinearOrder 𝕜] [OrderTopology 𝕜] [
    CompleteSpace 𝕜]
  [AddCommGroup E] [TopologicalSpace E] [IsTopologicalAddGroup E] [T2Space E]
  [Module 𝕜 E] [ContinuousSMul 𝕜 E] [FiniteDimensional 𝕜 E]
  [PseudoMetricSpace F] [ProperSpace F] [AddCommGroup F] [IsTopologicalAddGroup F] [T2Space F]
  [Module 𝕜 F] [ContinuousSMul 𝕜 F] [FiniteDimensional 𝕜 F]
  (f : E →ₗ[𝕜] F →ₗ[𝕜] 𝕜) {S : Set F} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨆ y : S, f x y
```

theorem continuous_iInf' [source]

A specialization of `LinearMap.continuous_iInf` to finite dimensional spaces, in place of requiring a (non-instance) `IsModuleTopology`.

```
theorem continuous_iInf' {E F 𝕜 : Type*}
  [NontriviallyNormedField 𝕜] [ConditionallyCompleteLinearOrder 𝕜] [OrderTopology 𝕜] [
    CompleteSpace 𝕜]
  [AddCommGroup E] [TopologicalSpace E] [IsTopologicalAddGroup E] [T2Space E]
  [Module 𝕜 E] [ContinuousSMul 𝕜 E] [FiniteDimensional 𝕜 E]
  [PseudoMetricSpace F] [ProperSpace F] [AddCommGroup F] [IsTopologicalAddGroup F] [T2Space F]
  [Module 𝕜 F] [ContinuousSMul 𝕜 F] [FiniteDimensional 𝕜 F]
  (f : E →ₗ[𝕜] F →ₗ[𝕜] 𝕜) {S : Set F} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨅ y : S, f x y
```

theorem BilinForm.continuous_iSup [source]

Alias of `LinearMap.continuous_iSup'` that takes `LinearMap.BilinForm`.

```
theorem BilinForm.continuous_iSup {𝕜 E : Type*}
  [RCLike 𝕜] [ConditionallyCompleteLinearOrder 𝕜] [OrderTopology 𝕜]
  [NormedAddCommGroup E] [InnerProductSpace 𝕜 E] [FiniteDimensional 𝕜 E] [ProperSpace E]
  (f : LinearMap.BilinForm 𝕜 E) {S : Set E} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨆ y : S, f x y
```

theorem BilinForm.continuous_iInf [source]

Alias of `LinearMap.continuous_iInf'` that takes `LinearMap.BilinForm`.

```
theorem BilinForm.continuous_iInf {𝕜 E : Type*}
  [RCLike 𝕜] [ConditionallyCompleteLinearOrder 𝕜] [OrderTopology 𝕜]
```

```
  [NormedAddCommGroup E] [InnerProductSpace 𝕜 E] [FiniteDimensional 𝕜 E] [ProperSpace E]
  (f : LinearMap.BilinForm 𝕜 E) {S : Set E} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨆ y : S, f x y
```

theorem ContinuousLinearMap.continuous_iSup [source]

A specialization of `Bornology.IsBounded.continuous_iSup_bilinear` *to* `ContinuousLinearMap`.

```
theorem continuous_iSup
  (f : F →SL[σ₁₂] E →L[𝕜] G) {S : Set E} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨆ y : S, f x y
```

theorem ContinuousLinearMap.continuous_iInf [source]

```
theorem continuous_iInf
  (f : F →SL[σ₁₂] E →L[𝕜] G) {S : Set E} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨅ y : S, f x y
```

theorem continuous_iSup_fst [source]

```
theorem LinearMap.BilinForm.continuous_iSup_fst
  {E : Type*} [NormedAddCommGroup E] [InnerProductSpace ℝ E] [FiniteDimensional ℝ E]
  (f : LinearMap.BilinForm ℝ E) {S : Set E} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨆ y : S, f y x
```

theorem continuous_iInf_fst [source]

```
theorem LinearMap.BilinForm.continuous_iInf_fst
  {E : Type*} [NormedAddCommGroup E] [InnerProductSpace ℝ E] [FiniteDimensional ℝ E]
  (f : LinearMap.BilinForm ℝ E) {S : Set E} (hS : Bornology.IsBounded S) :
    Continuous fun x ↦ ⨅ y : S, f y x
```

2.54 QuantumInfo.ForMathlib.Filter

QuantumInfo/ForMathlib/Filter.lean on GitHub.

theorem Tendsto_inv_nat_mul_div_real [source]

```
theorem Filter.Tendsto_inv_nat_mul_div_real (m : ℕ)
   : Filter.Tendsto (fun (x : ℕ) => ((↑x)⁻¹ * ↑(x / m) : ℝ)) Filter.atTop (𝓝 (1 / ↑m))
```

theorem tendsto_toReal_iff_of_eventually_ne_top [source]

```
theorem ENNReal.tendsto_toReal_iff_of_eventually_ne_top
  {ι} {fi : Filter ι} {f : ι → ℝ≥0∞} (hf : ∀ᶠ i in fi, f i ≠ ∞) {x : ℝ≥0∞}
    (hx : x ≠ ∞) : Tendsto (fun n => (f n).toReal) fi (𝓝 x.toReal) ↔ Tendsto f fi (𝓝 x)
```

2.55 QuantumInfo.ForMathlib.HermitianMat.Basic

QuantumInfo/ForMathlib/HermitianMat/Basic.lean on GitHub.

def HermitianMat [source]

The type of Hermitian matrices, as a `Subtype`. Equivalent to a `Matrix n n α` bundled with the fact that `Matrix.IsHermitian`.

```
def HermitianMat (n : Type*) (α : Type*) [AddGroup α] [StarAddMonoid α]
```

theorem eq_IsHermitian [source]

```
theorem eq_IsHermitian : HermitianMat n α  = { m : Matrix n n α // m.IsHermitian}
```

instance Coe (HermitianMat n α) (Matrix n n α) [source]

```
instance : Coe (HermitianMat n α) (Matrix n n α)
```

theorem val_eq_coe [source]

```
@[simp]
theorem val_eq_coe (A : HermitianMat n α) : A.val = A
```

theorem mat_mk [source]

```
@[simp]
theorem mat_mk (x : Matrix n n α) (h) : mat ⟨x, h⟩ = x
```

theorem mk_mat [source]

```
@[simp]
theorem mk_mat {A : HermitianMat n α} (h : A.mat.IsHermitian) : ⟨A.mat, h⟩ = A
```

theorem H [source]

Alias for HermitianMat.property or HermitianMat.2, this gets the fact that the value is actually `IsHermitian`.

```
theorem H (A : HermitianMat n α) : A.mat.IsHermitian
```

instance instFun [source]

```
instance instFun : FunLike (HermitianMat n α) n (n → α) where
```

theorem mat_apply [source]

```
@[simp]
theorem mat_apply {A : HermitianMat n α} {i j : n} : A.mat i j = A i j
```

theorem conjTranspose_mat [source]

```
@[simp]
theorem conjTranspose_mat (A : HermitianMat n α) :
    A.mat.conjTranspose = A.mat
```

instance AddGroup (HermitianMat n α) [source]

```
instance : AddGroup (HermitianMat n α)
```

instance Unique (HermitianMat n α) [source]

```
instance [IsEmpty n] : Unique (HermitianMat n α) where
```

theorem mat_zero [source]

```
@[simp, norm_cast]
theorem mat_zero : (0 : HermitianMat n α).mat = 0
```

theorem mk_zero [source]

```
@[simp]
theorem mk_zero (h : (0 : Matrix n n α).IsHermitian) : ⟨0, h⟩ = (0 : HermitianMat n α)
```

theorem zero_apply [source]

```
@[simp]
theorem zero_apply (i j : n) : (0 : HermitianMat n 𝕜) i j = 0
```

theorem mat_add [source]

```
@[simp, norm_cast]
theorem mat_add (A B : HermitianMat n α) :
    (A + B).mat = A.mat + B.mat
```

theorem mat_sub [source]

```
@[simp, norm_cast]
theorem mat_sub (A B : HermitianMat n α) :
    (A - B).mat = A.mat - B.mat
```

theorem mat_neg [source]

```
@[simp, norm_cast]
theorem mat_neg (A : HermitianMat n α) :
    (-A).mat = -A.mat
```

instance SMul R (HermitianMat n α) [source]

```
instance : SMul R (HermitianMat n α)
```

theorem mat_smul [source]

```
@[simp, norm_cast]
theorem mat_smul (c : R) (A : HermitianMat n α) :
    (c • A).mat = c • A.mat
```

theorem smul_apply [source]

```
@[simp]
theorem smul_apply (c : R) (A : HermitianMat n α) (i j : n) :
    (c • A) i j = c • A i j
```

instance TopologicalSpace (HermitianMat n α) [source]

```
instance : TopologicalSpace (HermitianMat n α)
```

theorem continuous_mat [source]

```
@[fun_prop]
theorem continuous_mat : Continuous (HermitianMat.mat : HermitianMat n α → Matrix n n α)
```

lemma continuousOn_iff_coe [source]

```
lemma continuousOn_iff_coe {X : Type*} [TopologicalSpace X] {s : Set X}
    (f : X → HermitianMat n α) :
    ContinuousOn f s ↔ ContinuousOn (fun x => (f x).mat) s
```

instance ContinuousAdd (HermitianMat n α) [source]

```
instance : ContinuousAdd (HermitianMat n α)
```

instance ContinuousNeg (HermitianMat n α) [source]

```
instance : ContinuousNeg (HermitianMat n α)
```

instance IsTopologicalAddGroup (HermitianMat n α) [source]

```
instance : IsTopologicalAddGroup (HermitianMat n α) where
```

instance ContinuousSMul R (HermitianMat n α) [source]

```
instance : ContinuousSMul R (HermitianMat n α) where
```

instance IsTopologicalAddGroup (HermitianMat n 𝕜) [source]

```
instance : IsTopologicalAddGroup (HermitianMat n 𝕜)
```

instance ContinuousSMul ℝ (HermitianMat n ℂ) [source]

```
instance : ContinuousSMul ℝ (HermitianMat n ℂ)
```

instance T3Space (HermitianMat n 𝕜) [source]

```
instance : T3Space (HermitianMat n 𝕜)
```

instance MulAction R (HermitianMat n α) [source]

```
instance : MulAction R (HermitianMat n α)
```

instance AddCommGroup (HermitianMat n α) [source]

```
instance : AddCommGroup (HermitianMat n α)
```

theorem mat_finset_sum [source]

```
@[simp, norm_cast]
theorem mat_finset_sum (f : ι → HermitianMat n α) (s : Finset ι) :
    (∑ i ∈ s, f i).mat = ∑ i ∈ s, (f i).mat
```

instance Module R (HermitianMat n α) [source]

```
instance : Module R (HermitianMat n α)
```

def mat$_l$ [source]

The projection from HermitianMat to Matrix, as a continuous linear map.

```
@[simps]
def matₗ : HermitianMat n α →L[R] Matrix n n α where
```

instance One (HermitianMat n α) [source]

```
instance : One (HermitianMat n α)
```

theorem mat_one [source]

```
@[simp, norm_cast]
theorem mat_one : (1 : HermitianMat n α).mat = 1
```

theorem mk_one [source]

```
@[simp]
theorem mk_one (h : (1 : Matrix n n α).IsHermitian) : ⟨1, h⟩ = (1 : HermitianMat n α)
```

theorem one_apply [source]

```
@[simp]
theorem one_apply (i j : n) : (1 : HermitianMat n α) i j = (1 : Matrix n n α) i j
```

instance AddCommMonoidWithOne (HermitianMat n 𝕜) [source]

```
noncomputable instance : AddCommMonoidWithOne (HermitianMat n 𝕜) where
```

instance NeZero (1 : HermitianMat n 𝕜) [source]

```
instance [i : Nonempty n] : NeZero (1 : HermitianMat n 𝕜)
```

instance instInv [source]

```
noncomputable instance instInv : Inv (HermitianMat m α)
```

theorem mat_inv [source]

```
@[simp, norm_cast]
theorem mat_inv : (A⁻¹).mat = A.mat⁻¹
```

theorem zero_inv [source]

```
@[simp]
theorem zero_inv : ((0 : HermitianMat m α)⁻¹) = 0
```

theorem one_inv [source]

```
@[simp]
theorem one_inv : ((1 : HermitianMat m α)⁻¹) = 1
```

instance instPow [source]

```
noncomputable instance instPow : Pow (HermitianMat m α) ℕ
```

theorem mat_pow [source]

```
@[simp, norm_cast]
theorem mat_pow (n : ℕ) : (A ^ n).mat = A.mat ^ n
```

theorem pow_zero [source]

```
@[simp]
theorem pow_zero : A ^ 0 = 1
```

theorem zero_pow [source]

```
@[simp]
theorem zero_pow (hn : n ≠ 0): (0 : HermitianMat m α) ^ n = 0
```

theorem one_pow [source]

```
@[simp]
theorem one_pow : ((1 : HermitianMat m α) ^ n) = 1
```

instance instZPow [source]

```
noncomputable instance instZPow : Pow (HermitianMat m α) ℤ
```

theorem mat_zpow [source]

```
@[simp]
theorem mat_zpow (z : ℤ) : (A ^ z).mat = A.mat ^ z
```

theorem zpow_natCast [source]

```
@[simp, norm_cast]
theorem zpow_natCast : A ^ (n : ℤ) = A ^ n
```

theorem zpow_zero [source]

```
@[simp]
theorem zpow_zero : A ^ (0 : ℤ) = 1
```

theorem zpow_one [source]

```
@[simp]
theorem zpow_one : A ^ (1 : ℤ) = A
```

theorem one_zpow [source]

```
@[simp]
theorem one_zpow : ((1 : HermitianMat m α) ^ z) = 1
```

theorem zpow_neg_one [source]

```
@[simp]
theorem zpow_neg_one : A ^ (-1) = A⁻¹
```

theorem inv_zpow [source]

```
@[simp]
theorem inv_zpow : A⁻¹ ^ z = (A ^ z)⁻¹
```

theorem inv_commute [source]

```
@[aesop safe apply (rule_sets := [Commutes])]
theorem _root_.Matrix.inv_commute {α : Type*} {A : Matrix m m α} [CommRing α] : Commute A⁻¹ A
```

theorem commute_inv_self [source]

```
@[aesop safe apply (rule_sets := [Commutes])]
theorem commute_inv_self : Commute A⁻¹.mat A.mat
```

theorem commute_self_inv [source]

```
@[aesop safe apply (rule_sets := [Commutes])]
theorem commute_self_inv : Commute A.mat A⁻¹.mat
```

instance FiniteDimensional [source]

```
instance FiniteDimensional : FiniteDimensional ℝ (HermitianMat n 𝕜)
```

theorem im_diag_eq_zero [source]

```
@[simp]
theorem im_diag_eq_zero (A : HermitianMat n 𝕜) (x : n) :
    RCLike.im (A x x) = 0
```

theorem complex_im_eq_zero [source]

```
@[simp]
theorem complex_im_eq_zero (A : HermitianMat n ℂ) (x : n) :
    (A x x).im = 0
```

def conj [source]

The Hermitian matrix given by conjugating by a (possibly rectangular) Matrix. If we required `B` to be square, this would apply to any `Semigroup+StarMul` (as proved by `IsSelfAdjoint.conjugate`). But this lets us conjugate to other sizes too, as is done in e.g. Kraus operators. That is, it's a _heterogeneous_ conjguation.

```
def conj {m} (B : Matrix m n α) : HermitianMat n α →+ HermitianMat m α where
```

theorem conj_apply [source]

```
theorem conj_apply (B : Matrix m n α) (A : HermitianMat n α) :
    conj B A = ⟨B * A.mat * B.conjTranspose, (conj B A).2⟩
```

theorem conj_apply_mat [source]

```
@[simp]
theorem conj_apply_mat (B : Matrix m n α) (A : HermitianMat n α) :
    (A.conj B).mat = B * A.mat * B.conjTranspose
```

theorem conj_conj [source]

```
theorem conj_conj {m l} [Fintype m] (B : Matrix m n α) (C : Matrix l m α) :
    (A.conj B).conj C = A.conj (C * B)
```

theorem conj_zero [source]

```
@[simp]
theorem conj_zero [DecidableEq n] : A.conj (0 : Matrix m n α) = 0
```

theorem conj_one [source]

```
@[simp]
theorem conj_one [DecidableEq n] : A.conj 1 = A
```

lemma conj_one_unitary [source]

```
@[simp]
lemma conj_one_unitary [DecidableEq n] (U : Matrix.unitaryGroup n α) :
    conj U.val 1 = 1
```

def conjLinear [source]

HermitianMat.conj as an R-linear map, where R is the ring of relevant reals.

```
def conjLinear {m} (B : Matrix m n α) : HermitianMat n α →ₗ[R] HermitianMat m α where
```

theorem conjLinear_apply [source]

```
@[simp]
theorem conjLinear_apply (B : Matrix m n α) : conjLinear R B A = conj B A
```

lemma continuous_conj [source]

```
@[fun_prop]
lemma continuous_conj (ρ : HermitianMat n 𝕜) : Continuous (ρ.conj (m := m) ·)
```

instance FaithfulSMul ℝ (HermitianMat n 𝕜) [source]

```
instance [i : Nonempty n] : FaithfulSMul ℝ (HermitianMat n 𝕜) where
```

def lin [source]

The continuous linear map associated with a Hermitian matrix.

```
def lin : EuclideanSpace 𝕜 n →L[𝕜] EuclideanSpace 𝕜 n where
```

theorem isSymmetric [source]

```
@[simp]
theorem isSymmetric : A.lin.IsSymmetric
```

theorem lin_zero [source]

```
@[simp]
theorem lin_zero : (0 : HermitianMat n 𝕜).lin = 0
```

theorem lin_one [source]

```
@[simp]
theorem lin_one : (1 : HermitianMat n 𝕜).lin = 1
```

def eigenspace [source]

```
noncomputable def eigenspace (μ : 𝕜) : Submodule 𝕜 (EuclideanSpace 𝕜 n)
```

def ker [source]

The kernel of a Hermitian matrix `A` as a submodule of Euclidean space, defined by `LinearMap.ker A.toMat.toEuclideanLin`. Equivalently, the zero-eigenspace.

```
def ker : Submodule 𝕜 (EuclideanSpace 𝕜 n)
```

theorem mem_ker_iff_mulVec_zero [source]

```
theorem mem_ker_iff_mulVec_zero (x : EuclideanSpace 𝕜 n) : x ∈ A.ker ↔ A.mat.mulVec x = 0
```

theorem ker_eq_eigenspace_zero [source]

The kernel of a Hermitian matrix is its zero eigenspace.

```
theorem ker_eq_eigenspace_zero : A.ker = A.eigenspace 0
```

theorem ker_zero [source]

```
@[simp]
theorem ker_zero : (0 : HermitianMat n 𝕜).ker = ⊤
```

theorem ker_one [source]

```
@[simp]
theorem ker_one : (1 : HermitianMat n 𝕜).ker = ⊥
```

theorem ker_pos_smul [source]

```
theorem ker_pos_smul {c : ℝ} (hc : c ≠ 0) : (c • A).ker = A.ker
```

def support [source]

The support of a Hermitian matrix `A` as a submodule of Euclidean space, defined by `LinearMap.range A.toMat.toEuclideanLin`. Equivalently, the sum of all nonzero eigenspaces.

```
def support : Submodule 𝕜 (EuclideanSpace 𝕜 n)
```

theorem support_eq_sup_eigenspace_nonzero [source]

The support of a Hermitian matrix is the sum of its nonzero eigenspaces.

```
theorem support_eq_sup_eigenspace_nonzero : A.support = ⨆ μ ≠ 0, A.eigenspace μ
```

theorem support_zero [source]

```
@[simp]
theorem support_zero : (0 : HermitianMat n 𝕜).support = ⊥
```

theorem support_one [source]

```
@[simp]
theorem support_one : (1 : HermitianMat n 𝕜).support = ⊤
```

theorem ker_orthogonal_eq_support [source]

```
@[simp]
theorem ker_orthogonal_eq_support : A.ker^⊥ = A.support
```

theorem support_orthogonal_eq_range [source]

```
@[simp]
theorem support_orthogonal_eq_range : A.support^⊥ = A.ker
```

def diagonal [source]

```
def diagonal (f : n → ℝ) : HermitianMat n 𝕜
```

theorem diagonal_mat [source]

```
@[simp]
theorem diagonal_mat : (diagonal 𝕜 f).mat = Matrix.diagonal (f · : n → 𝕜)
```

theorem diagonal_zero [source]

```
@[simp]
theorem diagonal_zero : (diagonal 𝕜 0) = (0 : HermitianMat n 𝕜)
```

theorem diagonal_one [source]

```
@[simp]
theorem diagonal_one : (diagonal 𝕜 1) = (1 : HermitianMat n 𝕜)
```

lemma diagonal_add [source]

```
lemma diagonal_add : diagonal 𝕜 (f + g) = diagonal 𝕜 f + diagonal 𝕜 g
```

lemma diagonal_add_apply [source]

```
lemma diagonal_add_apply : diagonal 𝕜 (fun x ↦ f x + g x) = diagonal 𝕜 f + diagonal 𝕜 g
```

lemma diagonal_sub [source]

```
lemma diagonal_sub : diagonal 𝕜 (f - g) = diagonal 𝕜 f - diagonal 𝕜 g
```

theorem diagonal_mul [source]

```
theorem diagonal_mul (c : ℝ) : diagonal 𝕜 (fun x ↦ c * f x) = c • diagonal 𝕜 f
```

theorem diagonal_conj_diagonal [source]

```
theorem diagonal_conj_diagonal [Fintype n] :
    (diagonal 𝕜 f).conj (diagonal 𝕜 g) = diagonal 𝕜 (fun i ↦ f i * (g i)^2)
```

lemma eq_conj_diagonal [source]

A Hermitian matrix is equal to its diagonalization conjugated by its eigenvector unitary matrix.

```
lemma eq_conj_diagonal [Fintype n] (A : HermitianMat n 𝕜) :
    A = (diagonal 𝕜 A.H.eigenvalues).conj A.H.eigenvectorUnitary
```

def kronecker [source]

The kronecker product of two HermitianMats, see `Matrix.kroneckerMap`.

```
def kronecker (A : HermitianMat m α) (B : HermitianMat n α) : HermitianMat (m × n) α where
```

theorem kronecker_mat [source]

```
@[simp, norm_cast]
theorem kronecker_mat (A : HermitianMat m α) (B : HermitianMat n α) :
    (A ⊗ₖ B).mat = A.mat ⊗ₖ B.mat
```

theorem zero_kronecker [source]

```
@[simp]
theorem zero_kronecker (A : HermitianMat m α) : (0 : HermitianMat n α) ⊗ₖ A = 0
```

theorem kronecker_zero [source]

```
@[simp]
theorem kronecker_zero (A : HermitianMat m α) : A ⊗ₖ (0 : HermitianMat n α) = 0
```

theorem kronecker_one_one [source]

```
@[simp]
theorem kronecker_one_one : (1 : HermitianMat m α) ⊗ₖ (1 : HermitianMat n α) = 1
```

theorem add_kronecker [source]

```
theorem add_kronecker : (A + B) ⊗ₖ C = A ⊗ₖ C + B ⊗ₖ C
```

theorem kronecker_add [source]

```
theorem kronecker_add : A ⊗ₖ (B + C) = A ⊗ₖ B + A ⊗ₖ C
```

lemma kronecker_diagonal [source]

```
lemma kronecker_diagonal [DecidableEq m] [DecidableEq n] (d₁ : m → ℝ) (d₂ : n → ℝ) :
    (diagonal 𝕜 d₁ ⊗ₖ diagonal 𝕜 d₂) = diagonal 𝕜 (fun (i : m × n) => d₁ i.1 * d₂ i.2)
```

theorem kron_commute [source]

$A \otimes_k B$ always commutes with $C \otimes_k D$ if the pairs commute.

```
@[aesop safe apply (rule_sets := [Commutes])]
theorem kron_commute [Fintype m] [Fintype n] [DecidableEq m] [DecidableEq n]
    {A C : HermitianMat m α} {B D : HermitianMat n α}
    (hAC : Commute A.mat C.mat) (hBD : Commute B.mat D.mat):
    Commute (A ⊗ₖ B).mat (C ⊗ₖ D).mat
```

theorem kron_id_commute_id_kro [source]

$A \otimes_k 1$ always commutes with $1 \otimes_k B$

```
@[aesop safe apply (rule_sets := [Commutes])]
theorem kron_id_commute_id_kro [Fintype m] [Fintype n] [DecidableEq m] [DecidableEq n]
    (A : HermitianMat m α) (B : HermitianMat n α) :
    Commute (A ⊗ₖ (1 : HermitianMat n α)).mat ((1 : HermitianMat m α) ⊗ₖ B).mat
```

lemma kronecker_conj [source]

```
lemma kronecker_conj [Fintype m] [Fintype n]
    (A : HermitianMat m α) (B : HermitianMat n α) (C : Matrix p m α) (D : Matrix q n α) :
    (A ⊗ₖ B).conj (C ⊗ₖ D) = (A.conj C) ⊗ₖ (B.conj D)
```

theorem range_le_ker_imp_zero [source]

```
theorem range_le_ker_imp_zero {A : HermitianMat d 𝕜}
    (h : LinearMap.range A.mat.toEuclideanLin ≤ LinearMap.ker A.mat.toEuclideanLin) : A = 0
```

theorem range_mul_conjTranspose_of_ker_le_ker [source]

If $\ker M \subseteq \ker A$, then range $(A\ M^H) =$ range A.

```
theorem _root_.Matrix.range_mul_conjTranspose_of_ker_le_ker {A : Matrix d d 𝕜} {M : Matrix d₂ d
    𝕜}
    (h : LinearMap.ker M.toEuclideanLin ≤ LinearMap.ker A.toEuclideanLin) :
    LinearMap.range (A * M.conjTranspose).toEuclideanLin = LinearMap.range A.toEuclideanLin
```

theorem conj_ne_zero [source]

```
theorem conj_ne_zero {A : HermitianMat d 𝕜} {M : Matrix d₂ d 𝕜} (hA : A ≠ 0)
    (h : LinearMap.ker M.toEuclideanLin ≤ A.ker) : A.conj M ≠ 0
```

theorem conj_ne_zero_iff [source]

```
theorem conj_ne_zero_iff {A : HermitianMat d 𝕜} {M : Matrix d₂ d 𝕜}
    (h : LinearMap.ker M.toEuclideanLin ≤ A.ker) : A.conj M ≠ 0 ↔ A ≠ 0
```

theorem IsHermitian.spectrum_rcLike [source]

```
theorem _root_.Matrix.IsHermitian.spectrum_rcLike {A : Matrix n n 𝕜} (hA : A.IsHermitian) :
    RCLike.ofReal '' spectrum ℝ A = spectrum 𝕜 A
```

theorem HermitianMat.spectrum_rcLike [source]

We fix a simp-normal form that, for HermitianMat, we always work in terms of the real spectrum.

```
@[simp]
theorem spectrum_rcLike (A : HermitianMat n 𝕜) :
    spectrum 𝕜 A.mat = RCLike.ofReal '' spectrum ℝ A.mat
```

theorem ne_zero_iff_ne_zero_spectrum [source]

```
theorem ne_zero_iff_ne_zero_spectrum (A : HermitianMat n 𝕜) :
    A ≠ 0 ↔ ∃ x ∈ spectrum ℝ A.mat, x ≠ 0
```

theorem spectrum_prod [source]

```
theorem spectrum_prod
  {A : HermitianMat m 𝕜} {B : HermitianMat n 𝕜} :
    spectrum ℝ (A ⊗ₖ B).mat = spectrum ℝ A.mat * spectrum ℝ B.mat
```

instance AddCommMonoid (HermitianMat d ℂ) [source]

```
noncomputable instance : AddCommMonoid (HermitianMat d ℂ)
```

2.56 QuantumInfo.ForMathlib.HermitianMat.CFC

QuantumInfo/ForMathlib/HermitianMat/CFC.lean on GitHub.

theorem isSelfAdjoint [source]

```
@[aesop safe apply (rule_sets := [CStarAlgebra])]
theorem isSelfAdjoint : IsSelfAdjoint A.mat
```

theorem continuousOn_finite [source]

```
@[fun_prop]
theorem continuousOn_finite {α β : Type*} (f : α → β) (S : Set α)
    [TopologicalSpace α] [TopologicalSpace β] [T1Space α] [Finite S] : ContinuousOn f S
```

theorem conjTranspose_cfc [source]

```
@[simp]
theorem conjTranspose_cfc : (cfc f A.mat).conjTranspose = cfc f A.mat
```

def cfc [source]

```
protected def cfc : HermitianMat d 𝕜
```

theorem cfc_eq [source]

```
theorem cfc_eq : A.cfc f = ⟨cfc f A.mat, cfc_predicate f A.mat⟩
```

theorem mat_cfc [source]

```
@[simp]
theorem mat_cfc : (A.cfc f).mat = _root_.cfc f A.mat
```

theorem cfc_eq_cfc_iff_eqOn [source]

```
theorem cfc_eq_cfc_iff_eqOn (f g : ℝ → ℝ) :
    A.cfc f = A.cfc g ↔ Set.EqOn f g (spectrum ℝ A.mat)
```

theorem cfc_congr [source]

```
@[gcongr]
nonrec theorem cfc_congr (hfg : Set.EqOn f g (spectrum ℝ A.mat)) :
    A.cfc f = A.cfc g
```

theorem cfc_congr_of_nonneg [source]

Version of `cfc_congr` specialized to PSD matrices.

```
nonrec theorem cfc_congr_of_nonneg (hA : 0 ≤ A) (hfg : Set.EqOn f g (Set.Ici 0)) :
    A.cfc f = A.cfc g
```

theorem cfc_congr_of_posDef [source]

Version of `cfc_congr` specialized to positive definite matrices.

```
nonrec theorem cfc_congr_of_posDef (hA : A.mat.PosDef) (hfg : Set.EqOn f g (Set.Ioi 0)) :
    A.cfc f = A.cfc g
```

theorem cfc_left [source]

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem _root_.Commute.cfc_left (hAB : Commute A.mat B.mat) :
    Commute (A.cfc f).mat B.mat
```

theorem cfc_right [source]

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem _root_.Commute.cfc_right (hAB : Commute A.mat B.mat) :
    Commute A.mat (B.cfc f).mat
```

theorem cfc_commute [source]

```
theorem cfc_commute (f g : ℝ → ℝ) (hAB : Commute A.mat B.mat) :
    Commute (A.cfc f).mat (B.cfc g).mat
```

theorem cfc_self_commute [source]

```
@[aesop safe apply (rule_sets := [Commutes])]
theorem cfc_self_commute (A : HermitianMat d 𝕜) (f g : ℝ → ℝ) :
    Commute (A.cfc f).mat (A.cfc g).mat
```

theorem cfc_reindex [source]

Reindexing a matrix commutes with applying the CFC.

```
@[simp]
theorem cfc_reindex (e : d ≃ d₂) : (A.reindex e).cfc f = (A.cfc f).reindex e
```

theorem spectrum_cfc_eq_image [source]

```
theorem spectrum_cfc_eq_image (A : HermitianMat d 𝕜) (f : ℝ → ℝ) :
    spectrum ℝ (A.cfc f).mat = f '' (spectrum ℝ A.mat)
```

theorem cfc_toMat_eq_sum_smul_proj [source]

Spectral decomposition of `A.cfc f` as a sum of scaled projections (matrix version).

```
theorem cfc_toMat_eq_sum_smul_proj : (A.cfc f).mat =
    ∑ i, f (A.H.eigenvalues i) • (A.H.eigenvectorUnitary.val * (Matrix.single i i 1) * A.H.
    eigenvectorUnitary.val.conjTranspose)
```

theorem cfc_eigenvalues [source]

```
theorem cfc_eigenvalues (A : HermitianMat d 𝕜) :
    ∃ (e : d ≃ d), (A.cfc f).H.eigenvalues = f ∘ A.H.eigenvalues ∘ e
```

theorem cfc_id [source]

```
@[simp]
nonrec theorem cfc_id : A.cfc id = A
```

theorem cfc_id' [source]

```
@[simp]
nonrec theorem cfc_id' : A.cfc (·) = A
```

theorem cfc_add [source]

```
nonrec theorem cfc_add : A.cfc (f + g) = A.cfc f + A.cfc g
```

theorem cfc_add_apply [source]

```
theorem cfc_add_apply : A.cfc (fun x ↦ f x + g x) = A.cfc f + A.cfc g
```

theorem cfc_sub [source]

```
nonrec theorem cfc_sub : A.cfc (f - g) = A.cfc f - A.cfc g
```

theorem cfc_sub_apply [source]

```
theorem cfc_sub_apply : A.cfc (fun x ↦ f x - g x) = A.cfc f - A.cfc g
```

theorem cfc_neg [source]

```
nonrec theorem cfc_neg : A.cfc (-f) = -A.cfc f
```

theorem cfc_neg_apply [source]

```
theorem cfc_neg_apply : A.cfc (fun x ↦ -f x) = -A.cfc f
```

theorem mat_cfc_mul [source]

We don't have a direct analog of `cfc_mul`, since we can't generally multiply to HermitianMat's to get another one, so the theorem statement wouldn't be well-typed. But, we can say that the matrices are always equal. See `cfc_conj` for the coe-free analog to multiplication.

```
theorem mat_cfc_mul : (A.cfc (f * g)).mat = A.cfc f * A.cfc g
```

theorem mat_cfc_mul_apply [source]

```
theorem mat_cfc_mul_apply : (A.cfc (fun x ↦ f x * g x)).mat = A.cfc f * A.cfc g
```

theorem cfc_comp [source]

```
nonrec theorem cfc_comp : A.cfc (g ∘ f) = (A.cfc f).cfc g
```

theorem cfc_comp_apply [source]

```
theorem cfc_comp_apply : A.cfc (fun x ↦ g (f x)) = (A.cfc f).cfc g
```

theorem cfc_conj [source]

```
nonrec theorem cfc_conj : (A.cfc f).conj (A.cfc g) = A.cfc (f * g^2)
```

theorem cfc_diagonal [source]

```
@[simp]
theorem cfc_diagonal (g : d → ℝ) : (diagonal 𝕜 g).cfc f = diagonal 𝕜 (f ∘ g)
```

theorem cfc_conj_unitary [source]

```
theorem cfc_conj_unitary (U : Matrix.unitaryGroup d 𝕜) :
    (A.conj U.val).cfc f = (A.cfc f).conj U
```

theorem cfc_const [source]

```
@[simp]
nonrec theorem cfc_const : (A.cfc (fun _ ↦ r)) = r • 1
```

theorem cfc_const_mul_id [source]

```
@[simp]
nonrec theorem cfc_const_mul_id : A.cfc (fun x ↦ r * x) = r • A
```

theorem cfc_const_mul [source]

```
@[simp]
nonrec theorem cfc_const_mul : A.cfc (fun x ↦ r * f x) = r • A.cfc f
```

theorem cfc_apply_zero [source]

```
@[simp]
nonrec theorem cfc_apply_zero : (0 : HermitianMat d 𝕜).cfc f = f 0 • 1
```

theorem cfc_apply_one [source]

```
@[simp]
nonrec theorem cfc_apply_one : (1 : HermitianMat d 𝕜).cfc f = f 1 • 1
```

theorem cfc_pow [source]

```
theorem cfc_pow {n : ℕ} : A.cfc (· ^ n) = A ^ n
```

theorem cfc_nonneg_iff [source]

```
theorem cfc_nonneg_iff : 0 ≤ A.cfc f ↔ ∀ i, 0 ≤ f (A.H.eigenvalues i)
```

theorem cfc_posDef [source]

```
theorem cfc_posDef : (A.cfc f).mat.PosDef ↔ ∀ i, 0 < f (A.H.eigenvalues i)
```

theorem cfc_nonneg_of_nonneg [source]

If a rael function preserves nonnegativity, the CFC preserves PSDness.

```
theorem cfc_nonneg_of_nonneg (hA : 0 ≤ A) (hf : ∀ i ≥ 0, 0 ≤ f i) :
    0 ≤ A.cfc f
```

theorem cfc_nonSingular [source]

```
theorem cfc_nonSingular (hf : ∀ i, f (A.H.eigenvalues i) ≠ 0) : NonSingular (A.cfc f)
```

theorem trace_mul_cfc [source]

```
theorem trace_mul_cfc (A : HermitianMat d 𝕜) (f : ℝ → ℝ) :
    (A.mat * (A.cfc f).mat).trace = ∑ i, A.H.eigenvalues i * f (A.H.eigenvalues i)
```

theorem norm_eq_sum_eigenvalues_sq [source]

```
theorem norm_eq_sum_eigenvalues_sq (A : HermitianMat d 𝕜) :
    ‖A‖ ^ 2 = ∑ i, (A.H.eigenvalues i)^2
```

theorem lt_smul_of_norm_lt [source]

```
theorem lt_smul_of_norm_lt {r : ℝ} (h : ‖A‖ ≤ r) : A ≤ r • 1
```

theorem ball_subset_Icc [source]

```
theorem ball_subset_Icc : Metric.ball A r ⊆ Set.Icc (A - r • 1) (A + r • 1)
```

theorem spectrum_subset_of_mem_Icc [source]

```
theorem spectrum_subset_of_mem_Icc (A B : HermitianMat d 𝕜) :
    ∃ a b, ∀ x, A ≤ x ∧ x ≤ B → spectrum ℝ x.mat ⊆ Set.Icc a b
```

theorem cfc_continuous [source]

```
@[fun_prop]
protected theorem cfc_continuous {f : ℝ → ℝ} (hf : Continuous f) :
    Continuous (HermitianMat.cfc · f : HermitianMat d ℂ → HermitianMat d ℂ)
```

theorem spectrum_subset_Ioi [source]

```
theorem Matrix.PosDef.spectrum_subset_Ioi {d 𝕜 : Type*} [Fintype d] [DecidableEq d] [RCLike 𝕜]
    {A : Matrix d d 𝕜} (hA : A.PosDef) : spectrum ℝ A ⊆ Set.Ioi 0
```

theorem continuous_cfc_fun [source]

If f is a continuous family of functions parameterized by x, then (fun x => A.cfc (f x)) is also continuous.

```
@[fun_prop]
theorem continuous_cfc_fun {f : X → ℝ → ℝ} (hf : ∀ i, Continuous (f · i)) :
    Continuous (fun x ↦ A.cfc (f x))
```

theorem continuousOn_cfc_fun [source]

ContinuousOn variant for when all the matrices (A x) have a spectrum in a set T, and f is continuous on a set S.

```
@[fun_prop]
theorem continuousOn_cfc_fun {T : Set ℝ}
  (hf : ∀ i ∈ T, ContinuousOn (f · i) S) (hA : spectrum ℝ A.mat ⊆ T) :
    ContinuousOn (fun x ↦ A.cfc (f x)) S
```

lemma norm_cfc_le_sqrt_card_mul_bound [source]

Bound the Frobenius norm of a functional calculus application.

```
lemma norm_cfc_le_sqrt_card_mul_bound {A : HermitianMat d ℂ} {f : ℝ → ℝ} {C : ℝ}
    (hC : 0 ≤ C) (hf : ∀ x ∈ spectrum ℝ A.mat, ‖f x‖ ≤ C) :
    ‖A.cfc f‖ ≤ Real.sqrt (Fintype.card d) * C
```

lemma norm_cfc_sub_cfc_le_sqrt_card [source]

```
lemma norm_cfc_sub_cfc_le_sqrt_card {A : HermitianMat d ℂ} {f g : ℝ → ℝ} :
    ‖A.cfc f - A.cfc g‖ ≤ Real.sqrt (Fintype.card d) * ⨆ x ∈ spectrum ℝ A.mat, ‖f x - g x‖
```

lemma norm_cfc_sub_le_of_sup_le [source]

```
lemma norm_cfc_sub_le_of_sup_le {A : HermitianMat d ℂ} {f g : ℝ → ℝ} {T : Set ℝ} {ε : ℝ}
    (hT : spectrum ℝ A.mat ⊆ T) (hε : 0 ≤ ε) (h_sup : ∀ x ∈ T, ‖f x - g x‖ ≤ ε) :
    ‖A.cfc f - A.cfc g‖ ≤ Real.sqrt (Fintype.card d) * ε
```

lemma dist_lt_of_continuous' [source]

If f is jointly continuous on $S \{}times T$ and T is compact, then $x \{}mapsto f(x, \{}cdot)$ is continuous into the space of bounded functions on T with the uniform norm.

```
lemma dist_lt_of_continuous' {X : Type*} [TopologicalSpace X]
    {f : X → ℝ → ℝ} {S : Set X} {T : Set ℝ}
    (hT : IsCompact T)
    (hf : ContinuousOn (fun (p : X × ℝ) ↦ f p.1 p.2) (S ×ˢ T))
    {x₀ : X} (hx₀ : x₀ ∈ S) {ε : ℝ} (hε : 0 < ε) :
    ∃ U ∈ nhds x₀, ∀ x ∈ U ∩ S, ∀ t ∈ T, ‖f x t - f x₀ t‖ < ε
```

lemma continuousOn_cfc_of_compact [source]

The functional calculus is continuous on matrices with spectrum in a compact set.

```
lemma continuousOn_cfc_of_compact {K : Set ℝ} {g : ℝ → ℝ} (hK : IsCompact K) (hg : ContinuousOn
     g K) :
    ContinuousOn (fun (A : HermitianMat d ℂ) ↦ A.cfc g) {A | spectrum ℝ A.mat ⊆ K}
```

theorem continuous_cfc_joint_compact [source]

```
theorem continuous_cfc_joint_compact {X d : Type*} [TopologicalSpace X] [Fintype d] [
    DecidableEq d]
  {f : X → ℝ → ℝ} {A : X → HermitianMat d ℂ} {S : Set X} {T : Set ℝ}
  (hT : IsCompact T)
  (hf : ContinuousOn (fun (p : X × ℝ) ↦ f p.1 p.2) (S ×ˢ T))
  (hA₁ : ∀ x ∈ S, spectrum ℝ (A x).mat ⊆ T)
  (hA₂ : ContinuousOn (fun x ↦ A x) S) :
    ContinuousOn (fun x ↦ (A x).cfc (f x)) S
```

lemma eigenvalue_norm_le [source]

```
lemma eigenvalue_norm_le (A : HermitianMat d ℂ) (i : d) :
    |A.H.eigenvalues i| ≤ ‖A‖
```

lemma spectrum_subset_closedBall [source]

The spectrum of a `HermitianMat` is contained in the closed ball of radius $\|A\|$ around 0.

```
lemma spectrum_subset_closedBall (A : HermitianMat d ℂ) :
    spectrum ℝ A.mat ⊆ Metric.closedBall (0 : ℝ) ‖A‖
```

lemma spectrum_subset_of_isOpen [source]

```
lemma spectrum_subset_of_isOpen (A₀ : HermitianMat d ℂ) (U : Set ℝ)
    (hU : IsOpen U) (hAU : spectrum ℝ A₀.mat ⊆ U) :
    ∀ᶠ B in nhds A₀, spectrum ℝ B.mat ⊆ U
```

lemma continuousWithinAt_cfc_of_continuousOn [source]

```
lemma continuousWithinAt_cfc_of_continuousOn {T : Set ℝ} {g : ℝ → ℝ}
    {A₀ : HermitianMat d ℂ}
    (hg : ContinuousOn g T) (hA₀ : spectrum ℝ A₀.mat ⊆ T) :
    ContinuousWithinAt (fun B ↦ B.cfc g) {B | spectrum ℝ B.mat ⊆ T} A₀
```

lemma dist_lt_of_continuous_spectrum [source]

```
lemma dist_lt_of_continuous_spectrum {X : Type*} [TopologicalSpace X]
    {f : X → ℝ → ℝ} {A : X → HermitianMat d ℂ} {S : Set X} {T : Set ℝ}
    (hf : ContinuousOn (fun (p : X × ℝ) ↦ f p.1 p.2) (S ×ˢ T))
    (hA₁ : ∀ x ∈ S, spectrum ℝ (A x).mat ⊆ T)
    (hA₂ : ContinuousOn (fun x ↦ A x) S)
    {x₀ : X} (hx₀ : x₀ ∈ S) {ε : ℝ} (hε : 0 < ε) :
    ∃ U ∈ nhds x₀, ∀ y ∈ U ∩ S, ∀ t ∈ spectrum ℝ (A y).mat, ‖f y t - f x₀ t‖ < ε
```

theorem continuous_cfc_joint [source]

```
@[fun_prop]
theorem continuous_cfc_joint {X d : Type*} [TopologicalSpace X] [Fintype d] [DecidableEq d]
  {f : X → ℝ → ℝ} {A : X → HermitianMat d ℂ} {S : Set X} {T : Set ℝ}
  (hf : ContinuousOn (fun (p : X × ℝ) ↦ f p.1 p.2) (S ×ˢ T))
  (hA₁ : ∀ x ∈ S, spectrum ℝ (A x).mat ⊆ T)
  (hA₂ : ContinuousOn (fun x ↦ A x) S) :
    ContinuousOn (fun x ↦ (A x).cfc (f x)) S
```

theorem continuousOn_cfc_fun_nonsingular [source]

Specialization of `continuousOn_cfc_fun` for nonsingular matrices.

```
@[fun_prop]
theorem continuousOn_cfc_fun_nonsingular {f : X → ℝ → ℝ} {S : Set X}
  (hf : ∀ i ≠ 0, ContinuousOn (f · i) S) [NonSingular A] :
    ContinuousOn (fun x ↦ A.cfc (f x)) S
```

theorem continuousOn_cfc_fun_nonneg [source]

Specialization of `continuousOn_cfc_fun` for positive semidefinite matrices.

```
@[fun_prop]
theorem continuousOn_cfc_fun_nonneg {f : X → ℝ → ℝ} {S : Set X}
  (hf : ∀ i ≥ 0, ContinuousOn (f · i) S) (hA : 0 ≤ A) :
    ContinuousOn (fun x ↦ A.cfc (f x)) S
```

theorem continuousOn_cfc_fun_posDef [source]

Specialization of `continuousOn_cfc_fun` for positive definite matrices.

```
@[fun_prop]
theorem continuousOn_cfc_fun_posDef {f : X → ℝ → ℝ} {S : Set X}
  (hf : ∀ i > 0, ContinuousOn (f · i) S) (hA : A.mat.PosDef) :
    ContinuousOn (fun x ↦ A.cfc (f x)) S
```

lemma inv_cfc_eq_cfc_inv [source]

The inverse of the CFC is the CFC of the inverse function.

```
lemma inv_cfc_eq_cfc_inv (hf : ∀ i, f (A.H.eigenvalues i) ≠ 0) :
    (A.cfc f)⁻¹ = A.cfc (fun u ↦ (f u)⁻¹)
```

theorem cfc_inv [source]

```
theorem cfc_inv [NonSingular A] : A.cfc (fun u ↦ u⁻¹) = A⁻¹
```

lemma integral_toMat [source]

The integral of a Hermitian matrix function commutes with `toMat`.

```
lemma integral_toMat (A : ℝ → HermitianMat d 𝕜) (T₁ T₂ : ℝ) {μ : Measure ℝ}
  (hA : IntervalIntegrable A μ T₁ T₂) :
    (∫ t in T₁..T₂, A t ∂μ).mat = ∫ t in T₁..T₂, (A t).mat ∂μ
```

lemma intervalIntegrable_sum_smul_const [source]

A sum of scaled constant matrices is integrable if the scalar functions are integrable.

```
lemma intervalIntegrable_sum_smul_const (T₁ T₂ : ℝ) {μ : Measure ℝ} (g : ℝ → d → ℝ)
    (P : d → Matrix d d 𝕜) (hg : ∀ i, IntervalIntegrable (fun t ↦ g t i) μ T₁ T₂) :
    IntervalIntegrable (fun t ↦ ∑ i, g t i • P i) μ T₁ T₂
```

lemma intervalIntegrable_toMat_iff [source]

A function to Hermitian matrices is integrable iff its matrix values are integrable.

```
lemma intervalIntegrable_toMat_iff (A : ℝ → HermitianMat d 𝕜) (T₁ T₂ : ℝ) {μ : Measure ℝ} :
    IntervalIntegrable (fun t ↦ (A t).mat) μ T₁ T₂ ↔ IntervalIntegrable A μ T₁ T₂
```

lemma integrable_cfc [source]

The CFC of an integrable function family is integrable.

```
lemma integrable_cfc (T₁ T₂ : ℝ) (f : ℝ → ℝ → ℝ) {μ : Measure ℝ}
    (hf : ∀ i, IntervalIntegrable (fun t ↦ f t (A.H.eigenvalues i)) μ T₁ T₂) :
    IntervalIntegrable (fun t ↦ A.cfc (f t)) μ T₁ T₂
```

lemma integral_cfc_eq_cfc_integral [source]

The integral of the CFC is the CFC of the integral.

```
lemma integral_cfc_eq_cfc_integral (T₁ T₂ : ℝ) {μ : Measure ℝ} (f : ℝ → ℝ → ℝ)
    (hf : ∀ i, IntervalIntegrable (fun t ↦ f t (A.H.eigenvalues i)) μ T₁ T₂) :
    ∫ t in T₁..T₂, A.cfc (f t) ∂μ = A.cfc (fun u ↦ ∫ t in T₁..T₂, f t u ∂μ)
```

theorem cfc_pos_of_pos [source]

```
theorem cfc_pos_of_pos {A : HermitianMat d 𝕜} {f : ℝ → ℝ} (hA : 0 < A)
    (hf : ∀ i > 0, 0 < f i) (hf₂ : 0 ≤ f 0) : 0 < A.cfc f
```

theorem exists_HermitianMat_cfc [source]

If two matrices A and B commute, then they is a common matrix with which they are both CFCs of. This is a variant of the common theorem that "commuting matrices can be simultaneously diagonalized."

```
theorem _root_.Commute.exists_HermitianMat_cfc (hAB : Commute A.mat B.mat) :
    ∃ C : HermitianMat d 𝕜, (∃ f : ℝ → ℝ, A = C.cfc f) ∧ (∃ g : ℝ → ℝ, B = C.cfc g)
```

theorem cfc_le_cfc_of_PosDef [source]

```
theorem cfc_le_cfc_of_PosDef (hfg : ∀ i, 0 < i → f i ≤ g i) (hA : A.mat.PosDef) :
    A.cfc f ≤ A.cfc g
```

theorem cfc_le_cfc_of_commute_monoOn [source]

```
theorem cfc_le_cfc_of_commute_monoOn (hf : MonotoneOn f (Set.Ioi 0))
  (hAB₁ : Commute A.mat B.mat) (hAB₂ : A ≤ B) (hA : A.mat.PosDef) (hB : B.mat.PosDef) :
    A.cfc f ≤ B.cfc f
```

theorem cfc_le_cfc_of_commute [source]

TODO: See above

```
theorem cfc_le_cfc_of_commute (hf : Monotone f) (hAB₁ : Commute A.mat B.mat) (hAB₂ : A ≤ B) :
    A.cfc f ≤ B.cfc f
```

theorem inv_ge_one_of_le_one [source]

```
theorem inv_ge_one_of_le_one (hA : A.mat.PosDef) (h : A ≤ 1) : 1 ≤ A⁻¹
```

lemma trace_cfc_eq [source]

The trace of cfc(f, A) equals the sum of f applied to eigenvalues.

```
lemma trace_cfc_eq (A : HermitianMat d ℂ) (f : ℝ → ℝ) :
    (A.cfc f).trace = ∑ i, f (A.H.eigenvalues i)
```

lemma mulVec_eq_zero_iff_inner_eigenvector_zero [source]

```
lemma mulVec_eq_zero_iff_inner_eigenvector_zero
    (A : HermitianMat d ℂ) (x : EuclideanSpace ℂ d) :
    A.mat.mulVec x = 0 ↔ ∀ i, A.H.eigenvalues i ≠ 0 → inner ℂ (A.H.eigenvectorBasis i) x = 0
```

lemma cfc_mulVec_expansion [source]

```
lemma cfc_mulVec_expansion (A : HermitianMat d ℂ) (f : ℝ → ℝ) (x : EuclideanSpace ℂ d) :
    (A.cfc f).mat.mulVec x = ∑ i, (f (A.H.eigenvalues i) : ℂ) • inner ℂ (A.H.eigenvectorBasis i
    ) x • A.H.eigenvectorBasis i
```

lemma ker_cfc_le_ker_on_set [source]

```
lemma ker_cfc_le_ker_on_set
    (hs : spectrum ℝ A.mat ⊆ s)
    (h : ∀ i ∈ s, f i = 0 → i = 0) :
    (A.cfc f).ker ≤ A.ker
```

lemma ker_cfc_le_ker [source]

```
lemma ker_cfc_le_ker (h : ∀ i, f i = 0 → i = 0) :
    (A.cfc f).ker ≤ A.ker
```

lemma ker_cfc_le_ker_nonneg [source]

```
lemma ker_cfc_le_ker_nonneg (hA : 0 ≤ A) (h : ∀ i ≥ 0, f i = 0 → i = 0) :
    (A.cfc f).ker ≤ A.ker
```

lemma ker_le_ker_cfc_on_set [source]

```
lemma ker_le_ker_cfc_on_set (hs : spectrum ℝ A.mat ⊆ s) (h : ∀ i ∈ s, i = 0 → f i = 0) :
    A.ker ≤ (A.cfc f).ker
```

lemma ker_le_ker_cfc [source]

```
lemma ker_le_ker_cfc (h : ∀ i, i = 0 → f i = 0) :
    A.ker ≤ (A.cfc f).ker
```

lemma ker_le_ker_cfc_nonneg [source]

```
lemma ker_le_ker_cfc_nonneg (hA : 0 ≤ A) (h : ∀ i ≥ 0, i = 0 → f i = 0) :
    A.ker ≤ (A.cfc f).ker
```

theorem ker_cfc_eq_ker [source]

```
theorem ker_cfc_eq_ker (h : ∀ i, f i = 0 ↔ i = 0) :
    (A.cfc f).ker = A.ker
```

theorem ker_cfc_eq_ker_nonneg [source]

```
theorem ker_cfc_eq_ker_nonneg (hA : 0 ≤ A) (h : ∀ i ≥ 0, f i = 0 ↔ i = 0) :
    (A.cfc f).ker = A.ker
```

2.57 QuantumInfo.ForMathlib.HermitianMat.Inner

QuantumInfo/ForMathlib/HermitianMat/Inner.lean on GitHub.

instance Inner R (HermitianMat n α) [source]

The Hermitian inner product, `Tr[AB]`. This is equal to `Matrix.trace (A * B)`, but gives real values when the matrices are complex, using `IsMaximalSelfAdjoint`.

```
instance : Inner R (HermitianMat n α) where
```

theorem inner_def [source]

```
theorem inner_def (A B : HermitianMat n α) :
    ⟪A, B⟫_R = selfadjMap (A.mat * B.mat).trace
```

theorem inner_add_right [source]

```
protected theorem inner_add_right : ⟪A, B + C⟫_R = ⟪A, B⟫_R + ⟪A, C⟫_R
```

theorem inner_add_left [source]

```
protected theorem inner_add_left : ⟪A + B, C⟫_R = ⟪A, C⟫_R + ⟪B, C⟫_R
```

theorem inner_zero_right [source]

```
@[simp]
protected theorem inner_zero_right : ⟪A, 0⟫_R = 0
```

theorem inner_zero_left [source]

```
@[simp]
protected theorem inner_zero_left : ⟪0, A⟫_R = 0
```

theorem inner_neg_left [source]

```
@[simp]
protected theorem inner_neg_left : ⟪-A, B⟫_R = -⟪A, B⟫_R
```

theorem inner_neg_right [source]

```
@[simp]
protected theorem inner_neg_right : ⟪A, -B⟫_R = -⟪A, B⟫_R
```

theorem inner_sub_left [source]

```
protected theorem inner_sub_left : ⟪A, B - C⟫_R = ⟪A, B⟫_R - ⟪A, C⟫_R
```

theorem inner_sub_right [source]

```
protected theorem inner_sub_right : ⟪A - B, C⟫_R = ⟪A, C⟫_R - ⟪B, C⟫_R
```

theorem inner_smul_left [source]

```
@[simp]
protected theorem inner_smul_left (r : R) : ⟪r • A, B⟫_R = r * ⟪A, B⟫_R
```

theorem inner_smul_right [source]

```
@[simp]
protected theorem inner_smul_right (r : R) : ⟪A, r • B⟫_R = r * ⟪A, B⟫_R
```

def inner_l [source]

The Hermitian inner product as bilinear form. Compare with `inner`$_l$ (in the root namespace) which requires an `InnerProductSpace` instance.

```
protected def inner_l : LinearMap.BilinForm R (HermitianMat n α) where
```

theorem inner_one [source]

```
@[simp]
theorem inner_one : ⟪A, 1⟫_R = A.trace
```

theorem one_inner [source]

```
@[simp]
theorem one_inner : ⟪1, A⟫_R = A.trace
```

theorem inner_eq_trace_mul [source]

The inner product for Hermtian matrices is equal to the trace of the product.

```
theorem inner_eq_trace_mul : algebraMap R α ⟪A, B⟫_R = (A.mat * B.mat).trace
```

theorem inner_comm [source]

```
theorem inner_comm : ⟪A, B⟫_R = ⟪B, A⟫_R
```

theorem inner_eq_trace_trivial [source]

`HermitianMat.inner` reduces to `Matrix.trace (A * B)` when the elements are a `TrivialStar`.

```
theorem inner_eq_trace_trivial : ⟪A, B⟫_α = (A.mat * B.mat).trace
```

theorem inner_eq_re_trace [source]

```
theorem inner_eq_re_trace : ⟪A, B⟫ = RCLike.re (A.mat * B.mat).trace
```

theorem inner_eq_trace_rc [source]

```
theorem inner_eq_trace_rc : ⟪A, B⟫ = (A.mat * B.mat).trace
```

theorem inner_self_nonneg [source]

```
theorem inner_self_nonneg: 0 ≤ ⟪A, A⟫
```

theorem inner_mul_nonneg [source]

```
theorem inner_mul_nonneg (h : 0 ≤ A.mat * B.mat) : 0 ≤ ⟪A, B⟫
```

theorem inner_ge_zero [source]

The inner product for PSD matrices is nonnegative.

```
theorem inner_ge_zero (hA : 0 ≤ A) (hB : 0 ≤ B) : 0 ≤ ⟪A, B⟫
```

theorem inner_mono [source]

```
theorem inner_mono (hA : 0 ≤ A) : B ≤ C → ⟪A, B⟫ ≤ ⟪A, C⟫
```

theorem inner_mono' [source]

```
theorem inner_mono' (hA : 0 ≤ A) : B ≤ C → ⟪B, A⟫ ≤ ⟪C, A⟫
```

theorem inner_le_mul_trace [source]

The inner product for PSD matrices is at most the product of their traces.

```
theorem inner_le_mul_trace (hA : 0 ≤ A) (hB : 0 ≤ B) : ⟪A, B⟫ ≤ A.trace * B.trace
```

theorem inner_zero_iff_aux_lemma [source]

```
private theorem inner_zero_iff_aux_lemma [DecidableEq n] (hA₁ : A.mat.PosSemidef) (hB₁ : B.mat.
    PosSemidef) :
  RCLike.re (A.val * B.val).trace = 0 ↔
    LinearMap.range (Matrix.toEuclideanLin A.val) ≤
      LinearMap.ker (Matrix.toEuclideanLin B.val)
```

theorem inner_zero_iff [source]

The inner product of two PSD matrices is zero iff they have disjoint support, i.e., each lives entirely in the other's kernel.

```
theorem inner_zero_iff [DecidableEq n] (hA₁ : 0 ≤ A) (hB₁ : 0 ≤ B)
    : ⟪A, B⟫ = 0 ↔ A.support ≤ B.ker
```

theorem reindex_inner [source]

```
@[simp]
theorem reindex_inner (e : d ≃ d₂) (B : HermitianMat d₂ 𝕜) :
    ⟪A.reindex e, B⟫ = ⟪A, B.reindex e.symm⟫
```

theorem inner_continuous [source]

```
@[fun_prop]
theorem inner_continuous : Continuous (Inner.inner ℝ (E := HermitianMat d 𝕜))
```

theorem inner_bilinForm_Continuous [source]

```
@[fun_prop]
theorem inner_bilinForm_Continuous (A : HermitianMat d 𝕜) : Continuous ⇑(HermitianMat.innerₗ A)
```

def InnerProductCore [source]

We define the Hermitian inner product as our "canonical" inner product, which does induce a norm. This disagrees slightly with Mathlib convention on the `Matrix` type, which avoids asserting one norm as there are several reasonable ones; for Hermitian matrices, though, this seem to be the right choice.

```
noncomputable def InnerProductCore : InnerProductSpace.Core ℝ (HermitianMat d 𝕜)
```

instance instNormedGroup [source]

The `HermitianMat` type inherits the Frobenius necessarily, since it's going to need the Hermitian inner product, and in Mathlib an `InnerProductSpace` always carries the corresponding norm.

```
noncomputable instance instNormedGroup : NormedAddCommGroup (HermitianMat d 𝕜)
```

theorem norm_eq_frobenius [source]

```
theorem norm_eq_frobenius (A : HermitianMat d 𝕜) :
    ‖A‖ = (∑ i : d, ∑ j : d, ‖A i j‖ ^ 2) ^ (1 / 2 : ℝ)
```

theorem norm_eq_sqrt_inner_self [source]

```
theorem norm_eq_sqrt_inner_self (A : HermitianMat d 𝕜) : ‖A‖ = √(⟪A, A⟫)
```

instance instNormedSpace [source]

```
noncomputable instance instNormedSpace : NormedSpace ℝ (HermitianMat d 𝕜) where
```

instance instInnerProductSpace [source]

```
noncomputable instance instInnerProductSpace : InnerProductSpace ℝ (HermitianMat d 𝕜)
```

instance CompleteSpace (HermitianMat d 𝕜) [source]

```
instance : CompleteSpace (HermitianMat d 𝕜)
```

instance NormedAddCommGroup (HermitianMat d ℝ) [source]

```
noncomputable instance : NormedAddCommGroup (HermitianMat d ℝ)
```

instance NormedAddCommGroup (HermitianMat d ℂ) [source]

```
noncomputable instance : NormedAddCommGroup (HermitianMat d ℂ)
```

def instOrderClosed [source]

```
def _root_.RCLike.instOrderClosed : OrderClosedTopology 𝕜 where
```

theorem dist_le_of_mem_Icc [source]

```
theorem dist_le_of_mem_Icc (x : HermitianMat d 𝕜) (hA : A ≤ x) (hB : x ≤ B) :
    ‖x - A‖ ≤ ‖B - A‖
```

theorem IsHermitian_isClosed [source]

```
theorem Matrix.IsHermitian_isClosed : IsClosed { A : Matrix n n 𝕜 | A.IsHermitian }
```

theorem PosSemiDef_isClosed [source]

```
theorem Matrix.PosSemiDef_isClosed : IsClosed { A : Matrix n n 𝕜 | A.PosSemidef }
```

theorem isClosed_nonneg [source]

```
theorem isClosed_nonneg : IsClosed { A : HermitianMat n 𝕜 | 0 ≤ A }
```

instance OrderClosedTopology (HermitianMat d 𝕜) [source]

```
instance : OrderClosedTopology (HermitianMat d 𝕜) where
```

instance CompactIccSpace (HermitianMat d 𝕜) [source]

Equivalently: the matrices `X` such that `X - A` is PSD and `B - X` is PSD, form a compact set.

```
instance : CompactIccSpace (HermitianMat d 𝕜) where
```

theorem unitInterval_IsCompact [source]

The PSD matrices that are `≤ 1` are a compact set. More generally, this is true of any closed interval, but stating that is a bit different because of how numerals are treated. The `0` and `1` here are already directly matrices, putting in an `(a : ℝ) • 1 ≤ m ∧ m ≤ (b : ℝ) • 1` involves casts. But that theorem should follow easily from this. More generally `A ≤ m ∧ m ≤ B` is compact.

```
theorem unitInterval_IsCompact : IsCompact {m : HermitianMat d 𝕜 | 0 ≤ m ∧ m ≤ 1}
```

theorem norm_one [source]

```
@[simp]
theorem norm_one : ‖(1 : HermitianMat d 𝕜)‖ = √(Fintype.card d : ℝ)
```

theorem norm_eq_trace_sq [source]

```
theorem norm_eq_trace_sq : ‖A‖ ^ 2 = (A.mat ^ 2).trace
```

lemma inner_eq_doubly_stochastic_sum [source]

The inner product ⟪A, B⟫ equals ∑_{ij} a_i b_j w_{ij} where a_i, b_j are eigenvalues and w_{ij} = ‖C_{ij}‖² for C = U_A^* U_B unitary.

```
lemma inner_eq_doubly_stochastic_sum {d : Type*} [Fintype d] [DecidableEq d]
    (A B : HermitianMat d ℂ) :
    let C
```

2.58 QuantumInfo.ForMathlib.HermitianMat.Jordan

QuantumInfo/ForMathlib/HermitianMat/Jordan.lean on GitHub.

def symmMul [source]

```
def symmMul : HermitianMat d 𝕜
```

theorem symmMul_comm [source]

```
theorem symmMul_comm : A.symmMul B = B.symmMul A
```

theorem symmMul_zero [source]

```
@[simp]
theorem symmMul_zero : A.symmMul 0 = 0
```

theorem zero_symmMul [source]

```
@[simp]
theorem zero_symmMul : symmMul 0 A = 0
```

theorem symmMul_toMat [source]

```
theorem symmMul_toMat : (A.symmMul B).mat =
    (2 : 𝕜)⁻¹ • (A.mat * B.mat + B.mat * A.mat)
```

theorem symmMul_of_commute [source]

```
@[simp]
theorem symmMul_of_commute (hAB : Commute A.mat B.mat) :
    (A.symmMul B).mat = A.mat * B.mat
```

theorem symmMul_self [source]

```
theorem symmMul_self : (symmMul A A).mat = A.mat * A.mat
```

theorem symmMul_one [source]

```
@[simp]
theorem symmMul_one : A.symmMul 1 = A
```

theorem one_symmMul [source]

```
@[simp]
theorem one_symmMul : symmMul 1 A = A
```

theorem symmMul_neg_one [source]

```
@[simp]
theorem symmMul_neg_one : A.symmMul (-1) = -A
```

theorem neg_one_symmMul [source]

```
@[simp]
theorem neg_one_symmMul : symmMul (-1) A = -A
```

instance CommMagma (HermitianMat d 𝕜) [source]

```
scoped instance : CommMagma (HermitianMat d 𝕜) where
```

theorem mul_eq_symmMul [source]

```
theorem mul_eq_symmMul : A * B = A.symmMul B
```

instance IsCommJordan (HermitianMat d 𝕜) [source]

```
scoped instance : IsCommJordan (HermitianMat d 𝕜) where
```

instance MulZeroClass (HermitianMat d 𝕜) [source]

```
scoped instance : MulZeroClass (HermitianMat d 𝕜) where
```

instance MulZeroOneClass (HermitianMat d 𝕜) [source]

```
scoped instance : MulZeroOneClass (HermitianMat d 𝕜) where
```

instance NonUnitalNonAssocRing (HermitianMat d 𝕜) [source]

```
scoped instance : NonUnitalNonAssocRing (HermitianMat d 𝕜) where
```

instance NonAssocRing (HermitianMat d 𝕜) [source]

```
scoped instance : NonAssocRing (HermitianMat d 𝕜) where
```

instance IsScalarTower ℝ (HermitianMat d 𝕜) (HermitianMat d 𝕜) [source]

```
scoped instance : IsScalarTower ℝ (HermitianMat d 𝕜) (HermitianMat d 𝕜) where
```

2.59 QuantumInfo.ForMathlib.HermitianMat.LogExp

QuantumInfo/ForMathlib/HermitianMat/LogExp.lean on GitHub.

theorem log_smul_of_ne_zero [source]

```
theorem Matrix.IsHermitian.log_smul_of_ne_zero {A : Matrix d d 𝕜} (hA : A.IsHermitian) (hx : x
    ≠ 0) :
    cfc Real.log (x • A) = (Real.log x) • cfc (if · = 0 then (0 : ℝ) else 1) A + cfc Real.log A
```

def exp [source]

Matrix exponential of a Hermitian matrix, as given by the continuous functional calculus with `Real.exp`

```
def exp (A : HermitianMat d 𝕜) : HermitianMat d 𝕜
```

theorem exp_left' [source]

Primed because `Commute.exp_left` refers to `NormedSpace.exp` instead of `HermitianMat.exp`.

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem _root_.Commute.exp_left' (hAB : Commute A.mat B.mat) :
    Commute (A.exp).mat B.mat
```

theorem exp_right' [source]

Primed because `Commute.exp_right` refers to `NormedSpace.exp` instead of `HermitianMat.exp`.

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem _root_.Commute.exp_right' (hAB : Commute A.mat B.mat) :
    Commute A.mat (B.exp).mat
```

theorem reindex_exp [source]

```
@[simp]
theorem reindex_exp (e : d ≃ d₂) : (A.reindex e).exp = A.exp.reindex e
```

instance nonSingular_exp [source]

```
instance nonSingular_exp : NonSingular A.exp
```

theorem exp_nonneg [source]

The matrix exponential of a Hermitian matrix is nonnegative.

```
theorem exp_nonneg (A : HermitianMat d 𝕜) : 0 ≤ A.exp
```

theorem exp_pos [source]

The matrix exponential of a Hermitian matrix is strictly positive (Loewner order). Requires `Nonempty` since over an empty index type every matrix equals zero and `0 < 0` is false.

```
theorem exp_pos [i : Nonempty d] (A : HermitianMat d 𝕜) : 0 < A.exp
```

def evalHermitianMatExp [source]

Positivity extension for `HermitianMat.exp`: always strictly positive if `Nonempty d`. TODO: We could add a fallback to give `nonnegative` if `Nonempty d` is not available, possibly also print a warning. (Users might often not have `Nonempty d` in context, and they probably want to.)

```
@[positivity HermitianMat.exp _]
def evalHermitianMatExp : PositivityExt where eval {_u _α} _zα _pα e
```

def log [source]

Matrix logarithm (base e) of a Hermitian matrix, as given by the elementwise real logarithm of the diagonal in a diagonalized form, using `Real.log`

*Note that this means that the nullspace of the image includes all of the nullspace of the original matrix. This contrasts to the standard definition, which is typically defined for positive *definite* matrices, and the nullspace of the image is exactly the (λ=1)-eigenspace of the original matrix. (We also get the (λ=-1)-eigenspace here!)*

It coincides with a standard definition if A is positive definite.

```
def log (A : HermitianMat d 𝕜) : HermitianMat d 𝕜
```

theorem log_left [source]

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem _root_.Commute.log_left (hAB : Commute A.mat B.mat) :
    Commute (A.log).mat B.mat
```

theorem log_right [source]

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem _root_.Commute.log_right (hAB : Commute A.mat B.mat) :
    Commute A.mat (B.log).mat
```

theorem reindex_log [source]

```
@[simp]
theorem reindex_log (e : d ≃ d₂) : (A.reindex e).log = A.log.reindex e
```

theorem log_zero [source]

```
@[simp]
theorem log_zero : (0 : HermitianMat d 𝕜).log = 0
```

theorem log_one [source]

```
@[simp]
theorem log_one : (1 : HermitianMat d 𝕜).log = 0
```

theorem log_smul_of_pos [source]

```
theorem log_smul_of_pos (A : HermitianMat d 𝕜) (hx : x ≠ 0) :
    (x • A).log = Real.log x • A.supportProj + A.log
```

theorem log_smul [source]

```
theorem log_smul {A : HermitianMat d 𝕜} {x : ℝ} (hx : x ≠ 0) [NonSingular A] :
    (x • A).log = Real.log x • 1 + A.log
```

theorem inv_antitone [source]

```
theorem inv_antitone (hA : A.mat.PosDef) (h : A ≤ B) : B⁻¹ ≤ A⁻¹
```

lemma integral_inv_sub_inv_finite [source]

```
lemma Real.integral_inv_sub_inv_finite (x T : ℝ) (hx : 0 < x) (hT : 0 < T) :
    ∫ t in (0)..T, (1 / (1 + t) - 1 / (x + t)) = Real.log x + Real.log ((1 + T) / (x + T))
```

lemma tendsto_log_div_add_atTop [source]

The limit of $\{}log((1+T)/(x+T))$ as $T \{}to \{}infty$ is 0, for $x > 0$.

```
lemma Real.tendsto_log_div_add_atTop (x : ℝ) :
    Filter.Tendsto (fun T => Real.log ((1 + T) / (x + T))) .atTop (nhds 0)
```

theorem logApprox_mono [source]

Monotonicity of the finite integral approximation of the logarithm.

```
theorem logApprox_mono {x y : HermitianMat d 𝕜} (hx : x.mat.PosDef) (hy : y.mat.PosDef)
    (hxy : x ≤ y) (T : ℝ) (hT : 0 < T) :
    ∫ t in (0)..T, ((1 + t)⁻¹ • (1 : HermitianMat d 𝕜) - (x + t • 1)⁻¹) ≤
    ∫ t in (0)..T, ((1 + t)⁻¹ • (1 : HermitianMat d 𝕜) - (y + t • 1)⁻¹)
```

def logApprox [source]

```
noncomputable def logApprox {n 𝕜 : Type*} [Fintype n] [DecidableEq n] [RCLike 𝕜]
    (x : HermitianMat n 𝕜) (T : ℝ) : HermitianMat n 𝕜
```

def scalarLogApprox [source]

```
noncomputable def scalarLogApprox (T : ℝ) (u : ℝ) : ℝ
```

theorem scalarLogApprox_eq [source]

```
theorem scalarLogApprox_eq (x T : ℝ) (hx : 0 < x) (hT : 0 < T) :
    scalarLogApprox T x = Real.log x + Real.log ((1 + T) / (x + T))
```

lemma integrand_eq [source]

The integrand in the log approximation is the CFC of the scalar integrand.

```
private lemma integrand_eq
    (x : HermitianMat d 𝕜) (hx : x.mat.PosDef) (t : ℝ) (ht : 0 ≤ t) :
    ((1 + t)⁻¹ • (1 : HermitianMat d 𝕜) - (x + t • 1)⁻¹) = x.cfc (fun u => (1 + t)⁻¹ - (u + t)
    ⁻¹)
```

theorem logApprox_eq_cfc_scalar [source]

The matrix log approximation is the CFC of the scalar log approximation.

```
theorem logApprox_eq_cfc_scalar
    (x : HermitianMat d 𝕜) (hx : x.mat.PosDef) (T : ℝ) (hT : 0 < T) :
    logApprox x T = x.cfc (scalarLogApprox T)
```

theorem logApprox_eq_log_add_error [source]

The log approximation is the log plus an error term.

```
theorem logApprox_eq_log_add_error
    (x : HermitianMat d 𝕜) (hx : x.mat.PosDef) (T : ℝ) (hT : 0 < T) :
    logApprox x T = x.log + x.cfc (fun u => Real.log ((1 + T) / (u + T)))
```

lemma tendsto_cfc_log_div_add_atTop [source]

The error term in the log approximation tends to 0 as T goes to infinity.

```
lemma tendsto_cfc_log_div_add_atTop (x : HermitianMat d 𝕜) :
    Tendsto (fun T => x.cfc (fun u => Real.log ((1 + T) / (u + T)))) atTop (nhds 0)
```

lemma tendsto_logApprox [source]

The log approximation converges to the matrix logarithm.

```
lemma tendsto_logApprox {x : HermitianMat d 𝕜} (hx : x.mat.PosDef) :
  Tendsto (fun T => logApprox x T) atTop (nhds x.log)
```

theorem posDef_of_posDef_le [source]

```
theorem posDef_of_posDef_le (hA : A.mat.PosDef) (hAB : A ≤ B) : B.mat.PosDef
```

theorem log_mono [source]

The matrix logarithm is operator monotone.

```
theorem log_mono (hA : A.mat.PosDef) (hAB : A ≤ B) : A.log ≤ B.log
```

theorem le_of_exp_commute [source]

Monotonicity of exp on commuting operators.

```
theorem le_of_exp_commute (hAB₂ : A.exp ≤ B.exp) :
    A ≤ B
```

lemma inv_convex [source]

The inverse function is operator convex on positive definite matrices.

```
lemma inv_convex {x y : HermitianMat d 𝕜} (hx : x.mat.PosDef) (hy : y.mat.PosDef)
    {⦃a b : ℝ⦄} (ha : 0 ≤ a) (hb : 0 ≤ b) (hab : a + b = 1) :
    (a • x + b • y)⁻¹ ≤ a • x⁻¹ + b • y⁻¹
```

lemma inv_shift_convex [source]

The shifted inverse function is operator convex on positive definite matrices.

```
lemma inv_shift_convex {x y : HermitianMat d 𝕜} (hx : x.mat.PosDef) (hy : y.mat.PosDef)
    {⦃a b : ℝ⦄} (ha : 0 ≤ a) (hb : 0 ≤ b) (hab : a + b = 1) (t : ℝ) (ht : 0 ≤ t) :
    (a • x + b • y + t • 1)⁻¹ ≤ a • (x + t • 1)⁻¹ + b • (y + t • 1)⁻¹
```

lemma integrable_inv_shift [source]

Definition of the approximation of the matrix logarithm.

```
lemma integrable_inv_shift {A : HermitianMat d 𝕜} (hA : A.mat.PosDef) (b : ℝ) (hb : 0 ≤ b) :
    IntervalIntegrable (fun t => (A + t • 1)⁻¹) volume 0 b
```

theorem logApprox_concave [source]

The finite integral approximation of the matrix logarithm is operator concave.

```
theorem logApprox_concave {n 𝕜 : Type*} [Fintype n] [DecidableEq n] [RCLike 𝕜]
    {x y : HermitianMat n 𝕜} (hx : x.mat.PosDef) (hy : y.mat.PosDef)
    {⦃a b : ℝ⦄} (ha : 0 ≤ a) (hb : 0 ≤ b) (hab : a + b = 1) (T : ℝ) (hT : 0 ≤ T) :
    a • x.logApprox T + b • y.logApprox T ≤ (a • x + b • y).logApprox T
```

theorem log_concave [source]

The matrix logarithm is operator concave.

```
theorem log_concave {x y : HermitianMat d 𝕜} (hx : x.mat.PosDef) (hy : y.mat.PosDef)
    {a b : ℝ} (ha : 0 ≤ a) (hb : 0 ≤ b) (hab : a + b = 1) :
    a • x.log + b • y.log ≤ (a • x + b • y).log
```

lemma log_kron_diagonal [source]

```
lemma log_kron_diagonal {m n 𝕜 : Type*} [Fintype m] [DecidableEq m] [Fintype n] [DecidableEq n]
     [RCLike 𝕜]
    {d₁ : m → ℝ} {d₂ : n → ℝ} (h₁ : ∀ i, 0 < d₁ i) (h₂ : ∀ j, 0 < d₂ j) :
    (diagonal 𝕜 d₁ ⊗ₖ diagonal 𝕜 d₂).log =
    (diagonal 𝕜 d₁).log ⊗ₖ 1 + 1 ⊗ₖ (diagonal 𝕜 d₂).log
```

lemma log_conj_unitary [source]

The logarithm of a Hermitian matrix conjugated by a unitary matrix is the conjugate of the logarithm.

```
lemma log_conj_unitary (A : HermitianMat d 𝕜) (U : Matrix.unitaryGroup d 𝕜) :
    (A.conj U.val).log = A.log.conj U.val
```

theorem inner_log_smul_of [source]

```
theorem inner_log_smul_of [NonSingular A] {x : ℝ} (hx : x ≠ 0) :
    ⟪(x • A).log, B⟫ = Real.log x * B.trace + ⟪A.log, B⟫
```

lemma log_kron_diagonal_with_proj [source]

```
lemma log_kron_diagonal_with_proj {f : d → ℝ} {g : d₂ → ℝ}  :
    (diagonal 𝕜 f ⊗ₖ diagonal 𝕜 g).log =
    (diagonal 𝕜 f).log ⊗ₖ (diagonal 𝕜 g).supportProj +
    (diagonal 𝕜 f).supportProj ⊗ₖ (diagonal 𝕜 g).log
```

lemma log_kron_with_proj [source]

Generalization of `HermitianMat.log_kron` for possibly singular matrices.

```
lemma log_kron_with_proj : (A ⊗ₖ B).log = A.log ⊗ₖ B.supportProj + A.supportProj ⊗ₖ B.log
```

theorem log_kron [source]

The matrix logarithm of the Kronecker product of two nonsingular Hermitian matrices is the sum of the Kronecker products of their logarithms with the identity matrix.

```
theorem log_kron [NonSingular A] [NonSingular B] : (A ⊗ₖ B).log = A.log ⊗ₖ 1 + 1 ⊗ₖ B.log
```

2.60 QuantumInfo.ForMathlib.HermitianMat.NonSingular

QuantumInfo/ForMathlib/HermitianMat/NonSingular.lean on GitHub.

theorem isUnit_smul [source]

```
theorem isUnit_smul {c : R} (hA : IsUnit c) {M : Matrix d d S} (hM : IsUnit M) :
    IsUnit (c • M : Matrix d d S)
```

theorem isUnit_natCast [source]

```
theorem isUnit_natCast {n : ℕ} (hn : n ≠ 0) [CharZero F] : IsUnit (n : Matrix d d F)
```

theorem isUnit_real_smul [source]

```
theorem isUnit_real_smul {r : ℝ} (hr : r ≠ 0) {M : Matrix d d 𝕜} (hM : IsUnit M) :
    IsUnit (r • M : Matrix d d 𝕜)
```

theorem isUnit_real_cast [source]

```
theorem isUnit_real_cast {r : ℝ} (hr : r ≠ 0) : IsUnit (r • 1 : Matrix d d 𝕜)
```

class NonSingular [source]

```
class NonSingular (A : HermitianMat n R) : Prop where
  isUnit : IsUnit A.mat
```

theorem isUnit_mat_of_nonSingular [source]

```
@[simp]
theorem isUnit_mat_of_nonSingular [NonSingular A] : IsUnit A.mat
```

theorem nonsingular_iff_isUnit [source]

```
theorem nonsingular_iff_isUnit : NonSingular A ↔ IsUnit A.mat
```

instance instHasInv_of_invertible [source]

```
instance instHasInv_of_invertible [i : Invertible A.mat] : NonSingular A
```

instance instInvertible_of_hasInv [source]

```
instance instInvertible_of_hasInv [h : NonSingular A] : Invertible A.mat
```

instance NonSingular (1 : HermitianMat n R) [source]

```
instance : NonSingular (1 : HermitianMat n R)
```

theorem nonSingular_of_posDef [source]

```
theorem nonSingular_of_posDef (hA : A.mat.PosDef) : NonSingular A
```

theorem nonSingular_iff_posDef_of_PSD [source]

```
theorem nonSingular_iff_posDef_of_PSD (hA : 0 ≤ A) :
    NonSingular A ↔ A.mat.PosDef
```

theorem nonSingular_smul [source]

```
theorem nonSingular_smul [i : NonSingular A] {c : ℝ} (hc : IsUnit c) : NonSingular (c • A)
```

theorem nonSingular_iff_zero_notMem_spectrum [source]

```
theorem nonSingular_iff_zero_notMem_spectrum : NonSingular A ↔ 0 ∉ spectrum ℝ A.mat
```

theorem nonSingular_iff_eigenvalue_ne_zero [source]

```
theorem nonSingular_iff_eigenvalue_ne_zero : NonSingular A ↔ ∀ i, A.H.eigenvalues i ≠ 0
```

theorem nonSingular_iff_det_ne_zero [source]

```
theorem nonSingular_iff_det_ne_zero : NonSingular A ↔ A.mat.det ≠ 0
```

theorem nonSingular_iff_ker_bot [source]

```
theorem nonSingular_iff_ker_bot : NonSingular A ↔ A.ker = ⊥
```

theorem nonSingular_iff_support_top [source]

```
theorem nonSingular_iff_support_top : NonSingular A ↔ A.support = ⊤
```

theorem nonSingular_iff_neg [source]

```
@[simp]
theorem nonSingular_iff_neg : NonSingular (-A) ↔ NonSingular A
```

theorem nonSingular_iff_inv [source]

```
@[simp]
theorem nonSingular_iff_inv : NonSingular (A⁻¹) ↔ NonSingular A
```

theorem nonSingular_iff_kronecker [source]

```
@[simp]
theorem nonSingular_iff_kronecker [Nonempty n] [Nonempty m] :
    NonSingular (A ⊗ₖ B) ↔ NonSingular A ∧ NonSingular B
```

theorem nonSingular_iff_conj [source]

```
theorem nonSingular_iff_conj (hC : IsUnit C) : NonSingular (A.conj C) ↔ NonSingular A
```

theorem nonSingular_iff_reindex [source]

```
@[simp]
theorem nonSingular_iff_reindex (e : n ≃ m) : NonSingular (A.reindex e) ↔ NonSingular A
```

theorem nonSingular_empty [source]

```
theorem nonSingular_empty [IsEmpty n] : NonSingular A
```

theorem nonSingular_det_ne_zero [source]

```
theorem nonSingular_det_ne_zero : A.mat.det ≠ 0
```

theorem nonSingular_ker_bot [source]

```
@[simp]
theorem nonSingular_ker_bot : A.ker = ⊥
```

theorem nonSingular_support_top [source]

```
@[simp]
theorem nonSingular_support_top : A.support = ⊤
```

instance nonSingular_neg [source]

```
instance nonSingular_neg : NonSingular (-A)
```

instance nonSingular_inv [source]

```
instance nonSingular_inv : NonSingular (A⁻¹)
```

instance nonSingular_kron [source]

```
instance nonSingular_kron [Nonempty n] [Nonempty m] : NonSingular (A ⊗ₖ B)
```

theorem nonSingular_conj [source]

```
theorem nonSingular_conj (hC : IsUnit C) : NonSingular (A.conj C)
```

instance nonSingular_conj_isometry [source]

```
instance nonSingular_conj_isometry {B : HermitianMat n 𝕜} [NonSingular B] :
    NonSingular (A.conj B.mat)
```

theorem nonSingular_zero_notMem_spectrum [source]

```
theorem nonSingular_zero_notMem_spectrum : 0 ∉ spectrum ℝ A.mat
```

theorem nonSingular_eigenvalue_ne_zero [source]

```
theorem nonSingular_eigenvalue_ne_zero : ∀ i, A.H.eigenvalues i ≠ 0
```

2.61 QuantumInfo.ForMathlib.HermitianMat.Order

QuantumInfo/ForMathlib/HermitianMat/Order.lean on GitHub.

instance PartialOrder (HermitianMat n 𝕜) [source]

The `MatrixOrder` instance for Matrix (the Loewner order) we keep open for HermitianMat, always.

```
instance : PartialOrder (HermitianMat n 𝕜)
```

instance IsOrderedAddMonoid (HermitianMat n 𝕜) [source]

```
instance : IsOrderedAddMonoid (HermitianMat n 𝕜)
```

theorem le_iff [source]

```
theorem le_iff : A ≤ B ↔ (B - A).mat.PosSemidef
```

theorem zero_le_iff [source]

```
theorem zero_le_iff : 0 ≤ A ↔ A.mat.PosSemidef
```

theorem le_iff_mulVec_le [source]

```
theorem le_iff_mulVec_le : A ≤ B ↔
    ∀ x, star x ⬝ᵥ A.mat *ᵥ x ≤ star x ⬝ᵥ B.mat *ᵥ x
```

instance ZeroLEOneClass (HermitianMat n 𝕜) [source]

```
instance [DecidableEq n] : ZeroLEOneClass (HermitianMat n 𝕜) where
```

theorem lt_iff_posdef [source]

```
theorem lt_iff_posdef : A < B ↔ (B - A).mat.PosSemidef ∧ A ≠ B
```

instance IsStrictOrderedModule ℝ (HermitianMat n 𝕜) [source]

```
instance : IsStrictOrderedModule ℝ (HermitianMat n 𝕜) where
```

theorem posSemidef_iff_spectrum_Ici [source]

```
theorem posSemidef_iff_spectrum_Ici [DecidableEq n] (A : HermitianMat n 𝕜) :
    0 ≤ A ↔ spectrum ℝ A.mat ⊆ Set.Ici 0
```

theorem posSemidef_iff_spectrum_nonneg [source]

```
theorem posSemidef_iff_spectrum_nonneg [DecidableEq n] (A : HermitianMat n 𝕜) :
    0 ≤ A ↔ ∀ x ∈ spectrum ℝ A.mat, 0 ≤ x
```

theorem trace_nonneg [source]

```
theorem trace_nonneg (hA : 0 ≤ A) : 0 ≤ A.trace
```

theorem trace_pos [source]

```
theorem trace_pos (hA : 0 < A) : 0 < A.trace
```

def evalHermitianMatTrace [source]

Positivity extension for `HermitianMat.trace`: nonneg when the matrix is nonneg, positive when the matrix is positive.

```
@[positivity HermitianMat.trace _]
def evalHermitianMatTrace : PositivityExt where eval {_u _α} _zα _pα e
```

instance PosSMulMono ℝ (HermitianMat n 𝕜) [source]

```
instance : PosSMulMono ℝ (HermitianMat n 𝕜)
```

instance SMulPosMono ℝ (HermitianMat n 𝕜) [source]

```
instance : SMulPosMono ℝ (HermitianMat n 𝕜)
```

instance PosSMulReflectLE ℝ (HermitianMat n 𝕜) [source]

```
instance : PosSMulReflectLE ℝ (HermitianMat n 𝕜)
```

theorem le_trace_smul_one [source]

```
theorem le_trace_smul_one [DecidableEq n] (hA : 0 ≤ A) : A ≤ A.trace • 1
```

theorem kronecker_nonneg [source]

The Kronecker product of two nonnegative Hermitian matrices is nonnegative.

```
theorem kronecker_nonneg {A : HermitianMat m 𝕜} (hA : 0 ≤ A) (hB : 0 ≤ B) : 0 ≤ A ⊗ₖ B
```

theorem kronecker_pos [source]

The Kronecker product of two positive Hermitian matrices is positive.

```
theorem kronecker_pos {A : HermitianMat m 𝕜} (hA : 0 < A) (hB : 0 < B) : 0 < A ⊗ₖ B
```

theorem posSemidef_to_nonneg [source]

```
theorem posSemidef_to_nonneg {A : Matrix n n 𝕜} (hA : A.PosSemidef) : 0 ≤ A
```

theorem posDef_to_pos [source]

```
theorem posDef_to_pos {A : Matrix n n 𝕜} (hA : A.PosDef) [Nonempty n] : 0 < A
```

def findMatrixPSDInExpr [source]

Given an expression `e` (a matrix) and a proof expression `p` whose type may be `Matrix.PosSemidef A`, `Matrix.PosDef A`, or `And P Q` (syntactically), attempt to find a proof of nonnegativity or positivity for `e`. Only syntactic matching on the head constant is used; `isDefEq` is used only to compare the matrix argument.

```
private partial def findMatrixPSDInExpr (e : Expr) (p : Expr) (ty : Expr) :
    MetaM (Option (Bool × Expr))
```

def evalMatrixPSD [source]

Positivity extension for `Matrix`: looks for `A.PosSemidef` or `A.PosDef` in the local context (including syntactic `And` conjunctions) to prove `0 ≤ A` or `0 < A`.

```
@[positivity (_ : HermitianMat _ _)]
def evalMatrixPSD : PositivityExt where eval {_u _α} _zα _pα e
```

theorem mat_posSemidef_to_nonneg [source]

```
theorem mat_posSemidef_to_nonneg (hA : A.mat.PosSemidef) : 0 ≤ A
```

theorem mat_posDef_to_pos [source]

```
theorem mat_posDef_to_pos [Nonempty n] (hA : A.mat.PosDef) : 0 < A
```

def findHermitianMatPSDInExpr [source]

Given an expression `e` (a `HermitianMat`) and a proof expression `p` whose type may be `Matrix.PosSemidef A.mat`, `Matrix.PosDef A.mat`, or `And P Q` (syntactically), attempt to find a proof of nonnegativity or positivity for `e`. Only syntactic matching on the head constant is used; `isDefEq` is used only to compare the `HermitianMat` argument.

```
private partial def findHermitianMatPSDInExpr (e : Expr) (p : Expr) (ty : Expr) :
    MetaM (Option (Bool × Expr))
```

def evalHermitianMatPSD [source]

Positivity extension for `HermitianMat`: looks for `A.mat.PosSemidef` or `A.mat.PosDef` in the local context (including syntactic `And` conjunctions) to prove `0 ≤ A` or `0 < A`.

```
@[positivity (_ : HermitianMat _ _)]
def evalHermitianMatPSD : PositivityExt where eval {_u _α} _zα _pα e
```

def evalHermitianMatKronecker [source]

Positivity extension for `HermitianMat.kronecker`: nonneg when both factors are.

```
@[positivity HermitianMat.kronecker _ _]
def evalHermitianMatKronecker : PositivityExt where eval {_u _α} _zα _pα e
```

theorem conj_nonneg [source]

Positivity extension for `HermitianMat.conj`: nonneg when the inner matrix is.

```
theorem conj_nonneg (hA : 0 ≤ A) : 0 ≤ A.conj M
```

theorem conj_pos [source]

```
theorem conj_pos [DecidableEq n] {A : HermitianMat n 𝕜} {M : Matrix m n 𝕜} (hA : 0 < A)
    (h : LinearMap.ker M.toEuclideanLin ≤ A.ker) : 0 < A.conj M
```

def evalHermitianMatConj [source]

Positivity extension for `HermitianMat.conj`: nonneg when the inner matrix is.

```
@[positivity HermitianMat.conj _ _]
def evalHermitianMatConj : PositivityExt where eval {_u _α} _zα _pα e
```

theorem convex_cone [source]

```
theorem convex_cone (hA : 0 ≤ A) (hB : 0 ≤ B) {c₁ c₂ : ℝ} (hc₁ : 0 ≤ c₁) (hc₂ : 0 ≤ c₂) :
    0 ≤ (c₁ • A + c₂ • B)
```

theorem sq_nonneg [source]

```
theorem sq_nonneg [DecidableEq n] : 0 ≤ A ^ 2
```

theorem ker_antitone [source]

```
theorem ker_antitone [DecidableEq n] (hA : 0 ≤ A) : A ≤ B → B.ker ≤ A.ker
```

theorem conj_mono [source]

```
theorem conj_mono (h : A ≤ B) : A.conj M ≤ B.conj M
```

lemma conj_posDef [source]

```
lemma conj_posDef [DecidableEq n] (hA : A.mat.PosDef) (hN : IsUnit N) :
    (A.conj N).mat.PosDef
```

lemma inv_conj [source]

```
lemma inv_conj [DecidableEq n] {M : Matrix n n 𝕜} (hM : IsUnit M) :
    (A.conj M)⁻¹ = A⁻¹.conj (M⁻¹)ᴴ
```

theorem le_iff_mulVec_le_mulVec [source]

```
theorem le_iff_mulVec_le_mulVec (A B : HermitianMat n 𝕜) :
    A ≤ B ↔ ∀ v : n → 𝕜, star v ⬝ᵥ A.mat *ᵥ v ≤ star v ⬝ᵥ B.mat *ᵥ v
```

theorem inner_mulVec_nonneg [source]

```
theorem inner_mulVec_nonneg (hA : 0 ≤ A) (v : n → 𝕜) :
    0 ≤ star v ⬝ᵥ A.mat *ᵥ v
```

theorem mem_ker_of_inner_mulVec_zero [source]

```
theorem mem_ker_of_inner_mulVec_zero [DecidableEq n] (hA : 0 ≤ A) (v : EuclideanSpace 𝕜 n)
    (h : star v ⬝ᵥ A.mat *ᵥ v = 0) : v ∈ A.ker
```

theorem ker_add [source]

```
theorem ker_add [DecidableEq n] (hA : 0 ≤ A) (hB : 0 ≤ B) :
    (A + B).ker = A.ker ⊓ B.ker
```

theorem ker_sum [source]

```
theorem ker_sum [DecidableEq n] (f : ι → HermitianMat n 𝕜) (hf : ∀ i, 0 ≤ f i) :
    (∑ i, f i).ker = ⨅ i, (f i).ker
```

theorem ker_conj [source]

```
theorem ker_conj [DecidableEq n] (hA : 0 ≤ A) (B : Matrix n n 𝕜) :
    (A.conj B).ker = Submodule.comap (Matrix.toEuclideanLin B.conjTranspose) A.ker
```

theorem ker_le_of_le_smul [source]

```
theorem ker_le_of_le_smul {α : ℝ} [DecidableEq n] (hα : α ≠ 0) (hA : 0 ≤ A) (hAB : A ≤ α • B) :
     B.ker ≤ A.ker
```

theorem HermitianMat.eigenvalues_nonneg [source]

If a HermitianMat is PSD, then its eigenvalues are nonneg.

```
theorem eigenvalues_nonneg [DecidableEq n] (hA : 0 ≤ A) (i : n) :
    0 ≤ A.H.eigenvalues i
```

theorem mat_nonneg [source]

If a HermitianMat is PSD, its underlying matrix is nonneg in the Loewner order.

```
theorem mat_nonneg (hA : 0 ≤ A) : 0 ≤ A.mat
```

theorem mat_pos [source]

If a HermitianMat is positive, its underlying matrix is positive in the Loewner order.

```
theorem mat_pos (hA : 0 < A) : 0 < A.mat
```

theorem nonneg_conjTranspose_mul_self [source]

*M^H * M is nonneg in the Loewner order, for any matrix M.*

```
theorem _root_.Matrix.nonneg_conjTranspose_mul_self {m : Type*} [Fintype m]
    (M : Matrix m n 𝕜) : 0 ≤ M.conjTranspose * M
```

theorem nonneg_self_mul_conjTranspose [source]

*M * M^H is nonneg in the Loewner order, for any matrix M.*

```
theorem _root_.Matrix.nonneg_self_mul_conjTranspose {m : Type*} [Fintype m]
    (M : Matrix n m 𝕜) : 0 ≤ M * M.conjTranspose
```

theorem subtype_mk_nonneg [source]

```
theorem subtype_mk_nonneg {M : Matrix m m 𝕜} (h : 0 ≤ M) :
    0 ≤ ((M, (Matrix.LE.le.posSemidef h).isHermitian) : HermitianMat m 𝕜)
```

theorem subtype_mk_pos [source]

```
theorem subtype_mk_pos {M : Matrix m m 𝕜} (h : 0 < M) :
    0 < ((M, (Matrix.LE.le.posSemidef h.le).isHermitian) : HermitianMat m 𝕜)
```

theorem Matrix.eigenvalues_nonneg [source]

```
private theorem _root_.Matrix.eigenvalues_nonneg [DecidableEq n] {M : Matrix n n 𝕜} (h : 0 ≤ M)
     (i : n) :
    0 ≤ (Matrix.LE.le.posSemidef h).isHermitian.eigenvalues i
```

def evalHermitianMatMat [source]

Positivity extension for A.mat where A : HermitianMat: nonneg when 0 ≤ A.

```
@[positivity HermitianMat.mat _]
def evalHermitianMatMat : PositivityExt where eval {_u _α} _zα _pα e
```

def evalHermitianMatVal [source]

Positivity extension for A.mat where A : HermitianMat: nonneg when 0 ≤ A.

```
@[positivity Subtype.val (_ : HermitianMat _ _)]
def evalHermitianMatVal : PositivityExt where eval {_u _α} _zα _pα e
```

def evalMatrixSelfMulConjTranspose [source]

*Positivity extension for M * M^H as a Matrix: always nonneg.*

```
@[positivity HMul.hMul _ (Matrix.conjTranspose _)]
def evalMatrixSelfMulConjTranspose : PositivityExt where eval {_u _α} _zα _pα e
```

def evalMatrixConjTransposeMulSelf [source]

Positivity extension for `M^H * M` as a Matrix: always nonneg.

```
@[positivity HMul.hMul (Matrix.conjTranspose _) _]
def evalMatrixConjTransposeMulSelf : PositivityExt where eval {_u _α} _zα _pα e
```

def evalHermitianMatMk [source]

Positivity extension for `(M, (pf : M.IsHermitian))` as a HermitianMat: equivalent to `0 ≤ M` in `MatrixOrder`.

```
@[positivity (Subtype.mk _ _ : HermitianMat _ _)]
def evalHermitianMatMk : PositivityExt where eval {_u _α} _zα _pα e
```

def evalMatrixEigenvalues [source]

Positivity extension for eigenvalues of a Matrix: `0 ≤ (_ : M.IsHermitian).eigenvalues i`. Will try to prove `0 ≤ M` in the `MatrixOrder`. If the proof is `A.H`, i.e. `M` comes from a HermitianMat, this will give `0 ≤ A.mat` which becomes `0 ≤ A` later.

```
@[positivity Matrix.IsHermitian.eigenvalues _ _]
def evalMatrixEigenvalues : PositivityExt where eval {_u _α} _zα _pα e
```

2.62 QuantumInfo.ForMathlib.HermitianMat.Proj

QuantumInfo/ForMathlib/HermitianMat/Proj.lean on GitHub.

def projector [source]

Given a Submodule (EuclideanSpace ...) to HermitianMat, this gives the projector onto that subspace, i.e. a matrix that squares to itself, preserves vectors in the submodule, and zeroes out anything in the orthogonal complement of that submodule.

```
noncomputable def projector (S : Submodule 𝕜 (EuclideanSpace 𝕜 n)) : HermitianMat n 𝕜
```

theorem projector_add_orthogonal [source]

```
theorem projector_add_orthogonal : projector S + projector S^⊥ = 1
```

theorem trace_projector [source]

```
@[simp]
theorem trace_projector : (projector S).trace = (Module.finrank 𝕜 S : ℝ)
```

def supportProj [source]

The `HermitianMat.projector` for the `HermitianMat.support` submodule.

```
noncomputable def supportProj (A : HermitianMat n 𝕜) : HermitianMat n 𝕜
```

def kerProj [source]

The `HermitianMat.projector` for the `HermitianMat.ker` submodule.

```
noncomputable def kerProj (A : HermitianMat n 𝕜) : HermitianMat n 𝕜
```

theorem kerProj_add_supportProj [source]

```
@[simp]
theorem kerProj_add_supportProj : A.kerProj + A.supportProj = 1
```

theorem kerProj_of_nonSingular [source]

```
@[simp]
theorem kerProj_of_nonSingular [NonSingular A] : A.kerProj = 0
```

theorem supportProj_of_nonSingular [source]

```
@[simp]
theorem supportProj_of_nonSingular [NonSingular A] : A.supportProj = 1
```

theorem projector_eq_sum_rankOne [source]

The projector onto a submodule S is the sum of the outer products of the vectors in an orthonormal basis of S.

```
theorem projector_eq_sum_rankOne (b : OrthonormalBasis ι 𝕜 S) :
    (projector S).mat = ∑ i, Matrix.vecMulVec (S.subtype (b i)) (star (S.subtype (b i)))
```

lemma projector_support_eq_sum [source]

The projector onto the support of A is the sum of the projections onto the eigenvectors with non-zero eigenvalues.

```
lemma projector_support_eq_sum : A.supportProj.mat =
    ∑ i, (if A.H.eigenvalues i = 0 then 0 else 1) •
      Matrix.vecMulVec (A.H.eigenvectorBasis i) (star (A.H.eigenvectorBasis i))
```

theorem supportProj_eq_cfc [source]

```
theorem supportProj_eq_cfc : A.supportProj = A.cfc (if · = 0 then 0 else 1)
```

def projLE [source]

Projector onto the non-negative eigenspace of `B - A`. Accessible by the notation `{A ≤ₚ B}`, which is scoped to `HermitianMat`. This is the unique maximum operator `P` such that `P^2 = P` and `P * A * P ≤ P * B * P` in the Loewner order.

```
def projLE (A B : HermitianMat n 𝕜) : HermitianMat n 𝕜
```

def projLT [source]

Projector onto the positive eigenspace of `B - A`. Accessible by the notation `{A <ₚ B}`, which is scoped to `HermitianMat`. Compare with `proj_le`.

```
noncomputable def projLT (A B : HermitianMat n 𝕜) : HermitianMat n 𝕜
```

theorem projLE_def [source]

```
theorem projLE_def : {A ≤ₚ B} = (B - A).cfc (fun x ↦ if 0 ≤ x then 1 else 0)
```

theorem projLT_def [source]

```
theorem projLT_def : {A <ₚ B} = (B - A).cfc (fun x ↦ if 0 < x then 1 else 0)
```

theorem projLE_sq [source]

```
theorem projLE_sq : {A ≤ₚ B}^2 = {A ≤ₚ B}
```

theorem projLT_sq [source]

```
theorem projLT_sq : {A <ₚ B}^2 = {A <ₚ B}
```

theorem projLE_zero_cfc [source]

```
theorem projLE_zero_cfc : {0 ≤ₚ A} = A.cfc (fun x ↦ if 0 ≤ x then 1 else 0)
```

theorem projLT_zero_cfc [source]

```
theorem projLT_zero_cfc : {0 <ₚ A} = A.cfc (fun x ↦ if 0 < x then 1 else 0)
```

theorem projLE_zero_cfc' [source]

```
theorem projLE_zero_cfc' : {A ≤ₚ 0} = A.cfc (fun x ↦ if x ≤ 0 then 1 else 0)
```

theorem projLT_zero_cfc' [source]

```
theorem projLT_zero_cfc' : {A <ₚ 0} = A.cfc (fun x ↦ if x < 0 then 1 else 0)
```

theorem projLE_nonneg [source]

```
theorem projLE_nonneg : 0 ≤ {A ≤ₚ B}
```

theorem projLT_nonneg [source]

```
theorem projLT_nonneg : 0 ≤ {A <ₚ B}
```

theorem projLE_le_one [source]

```
theorem projLE_le_one : {A ≤ₚ B} ≤ 1
```

theorem projLE_mul_nonneg [source]

```
theorem projLE_mul_nonneg : 0 ≤ {A ≤ₚ B}.mat * (B - A).mat
```

theorem projLE_mul_le [source]

```
theorem projLE_mul_le : {A ≤ₚ B}.mat * A.mat ≤ {A ≤ₚ B}.mat * B.mat
```

theorem proj_le_add_lt [source]

```
@[simp]
theorem proj_le_add_lt : {A <ₚ B} + {B ≤ₚ A} = 1
```

theorem conj_lt_add_conj_le [source]

```
theorem conj_lt_add_conj_le : A.conj {A <ₚ 0} + A.conj {0 ≤ₚ A} = A
```

theorem supportProj_eq_proj_lt_add_proj_lt [source]

```
theorem supportProj_eq_proj_lt_add_proj_lt (A : HermitianMat n 𝕜) :
    A.supportProj = {A <ₚ 0} + {0 <ₚ A}
```

instance PosPart (HermitianMat n 𝕜) [source]

The positive part of a Hermitian matrix: the projection onto its positive eigenvalues.

```
instance : PosPart (HermitianMat n 𝕜) where
```

instance NegPart (HermitianMat n 𝕜) [source]

The negative part of a Hermitian matrix: the projection onto its negative eigenvalues.

```
instance : NegPart (HermitianMat n 𝕜) where
```

theorem posPart_eq_cfc_max [source]

```
theorem posPart_eq_cfc_max : A⁺ = A.cfc (fun x ↦ x ⊔ 0)
```

theorem negPart_eq_cfc_min [source]

```
theorem negPart_eq_cfc_min : A⁻ = A.cfc (fun x ↦ -x ⊔ 0)
```

theorem posPart_eq_cfc_ite [source]

```
theorem posPart_eq_cfc_ite : A⁺ = A.cfc (fun x ↦ if 0 ≤ x then x else 0)
```

theorem negPart_eq_cfc_ite [source]

```
theorem negPart_eq_cfc_ite : A⁻ = A.cfc (fun x ↦ if x ≤ 0 then -x else 0)
```

theorem posPart_eq_posPart_toMat [source]

There is an existing (very slow) `PosPart` instance on `Matrix n n 𝕜`, this shows that this is equal.

```
theorem posPart_eq_posPart_toMat : A⁺ = A.mat⁺
```

theorem negPart_eq_negPart_toMat [source]

There is an existing (very slow) `PosPart` instance on `Matrix n n 𝕜`, this shows that this is equal.

```
theorem negPart_eq_negPart_toMat : A⁻ = A.mat⁻
```

theorem posPart_eq_cfc_lt [source]

The positive part can be equivalently described as the nonnegative part.

```
theorem posPart_eq_cfc_lt : A⁺ = A.cfc (fun x ↦ if 0 < x then x else 0)
```

theorem negPart_eq_cfc_lt [source]

The negative part can be equivalently described as the nonpositive part.

```
theorem negPart_eq_cfc_lt : A⁻ = A.cfc (fun x ↦ if x < 0 then -x else 0)
```

theorem posPart_add_negPart [source]

```
theorem posPart_add_negPart : A⁺ - A⁻ = A
```

theorem posPart_eq_self [source]

```
theorem posPart_eq_self {A : HermitianMat n 𝕜} (hA : 0 ≤ A) :
    A⁺ = A
```

theorem posPart_nonneg [source]

```
theorem posPart_nonneg : 0 ≤ A⁺
```

theorem negPart_nonneg [source]

```
theorem negPart_nonneg : 0 ≤ A⁻
```

theorem posPart_le [source]

```
theorem posPart_le : A ≤ A⁺
```

theorem posPart_mul_negPart [source]

```
theorem posPart_mul_negPart : A⁺.mat * A⁻.mat = 0
```

theorem projLE_inner_nonneg [source]

```
theorem projLE_inner_nonneg  : 0 ≤ ⟪{A ≤ₚ B}, (B - A)⟫
```

theorem projLE_inner_le [source]

```
theorem projLE_inner_le : ⟪{A ≤ₚ B}, A⟫ ≤ ⟪{A ≤ₚ B}, B⟫
```

theorem inner_projLE_nonneg [source]

```
theorem inner_projLE_nonneg : 0 ≤ ⟪{A ≤ₚ B}, (B - A)⟫
```

theorem inner_projLE_le [source]

```
theorem inner_projLE_le : ⟪{A ≤ₚ B}, A⟫ ≤ ⟪{A ≤ₚ B}, B⟫
```

theorem posPart_Continuous [source]

```
@[fun_prop]
theorem posPart_Continuous : Continuous (·⁺ : HermitianMat n ℂ → _)
```

theorem negPart_Continuous [source]

```
@[fun_prop]
theorem negPart_Continuous : Continuous (·⁻ : HermitianMat n ℂ → _)
```

theorem one_sub_projLT [source]

```
theorem one_sub_projLT : 1 - {B ≤ₚ A} = {A <ₚ B}
```

theorem projLT_mul_nonneg [source]

```
theorem projLT_mul_nonneg : 0 ≤ {A <ₚ B}.mat * (B - A).mat
```

theorem proj_lt_mul_lt [source]

```
theorem proj_lt_mul_lt : {A <ₚ B}.mat * A.mat ≤ {A <ₚ B}.mat * B.mat
```

theorem inner_negPart_nonpos [source]

```
theorem inner_negPart_nonpos : ⟪A, A⁻⟫ ≤ 0
```

theorem posPart_inner_negPart_zero [source]

```
@[simp]
theorem posPart_inner_negPart_zero : ⟪A⁺, A⁻⟫ = 0
```

theorem inner_negPart_zero_iff [source]

```
theorem inner_negPart_zero_iff : ⟪A, A⁻⟫ = 0 ↔ 0 ≤ A
```

theorem inner_negPart_neg_iff [source]

```
theorem inner_negPart_neg_iff : ⟪A, A⁻⟫ < 0 ↔ ¬0 ≤ A
```

theorem nonneg_iff_inner_nonneg [source]

The self-duality of the PSD cone: a matrix is PSD iff its inner product with all nonnegative matrices is non-negative.

```
theorem nonneg_iff_inner_nonneg (A : HermitianMat n 𝕜) :
    0 ≤ A ↔ ∀ B, 0 ≤ B → 0 ≤ ⟪A, B⟫
```

2.63 QuantumInfo.ForMathlib.HermitianMat.Reindex

QuantumInfo/ForMathlib/HermitianMat/Reindex.lean on GitHub.

def reindex [source]

```
def reindex (e : d ≃ d₂) : HermitianMat d₂ 𝕜
```

theorem mat_reindex [source]

```
@[simp]
theorem mat_reindex : (A.reindex e).mat = A.mat.reindex e e
```

theorem reindex_refl [source]

```
@[simp]
theorem reindex_refl (A : HermitianMat d 𝕜) :
    A.reindex (.refl _) = A
```

theorem reindex_reindex [source]

```
@[simp]
theorem reindex_reindex (A : HermitianMat d 𝕜) (e : d ≃ d₂) (f : d₂ ≃ d₃) :
    (A.reindex e).reindex f = A.reindex (e.trans f)
```

theorem reindex_zero [source]

```
@[simp]
theorem reindex_zero : (0 : HermitianMat d 𝕜).reindex e = 0
```

theorem reindex_one [source]

```
@[simp]
theorem reindex_one [DecidableEq d] [DecidableEq d₂] :
    (1 : HermitianMat d 𝕜).reindex e = 1
```

theorem reindex_add [source]

```
@[simp]
theorem reindex_add : A.reindex e + B.reindex e = (A + B).reindex e
```

theorem reindex_sub [source]

```
@[simp]
theorem reindex_sub  : A.reindex e - B.reindex e = (A - B).reindex e
```

theorem reindex_neg [source]

```
@[simp]
theorem reindex_neg : (-A).reindex e = -(A.reindex e)
```

theorem reindex_smul [source]

```
@[simp]
theorem reindex_smul (c : ℝ) : (c • A).reindex e = c • (A.reindex e)
```

theorem reindex_conj [source]

```
@[simp]
theorem reindex_conj [Fintype d₂] [Fintype d] (B : Matrix d₃ d₂ 𝕜) :
    (A.reindex e).conj B = A.conj (B.submatrix id e)
```

theorem conj_submatrix [source]

```
theorem conj_submatrix (B : Matrix d₂ d₄ 𝕜) (e : d₃ ≃ d₂) (f : d → d₄) :
    A.conj (B.submatrix e f) = (A.conj (B.submatrix id f)).reindex e.symm
```

theorem reindex_eq_conj [source]

```
theorem reindex_eq_conj [DecidableEq d] (e : d ≃ d₂) : A.reindex e = A.conj (Matrix.reindex e (.
    refl d) 1)
```

theorem ker_reindex [source]

```
theorem ker_reindex :
    (A.reindex e).ker = A.ker.comap (LinearEquiv.euclidean_of_relabel 𝕜 e).toLinearMap
```

theorem ker_reindex_le_iff [source]

```
@[simp]
theorem ker_reindex_le_iff :
    (A.reindex e).ker ≤ (B.reindex e).ker ↔ A.ker ≤ B.ker
```

2.64 QuantumInfo.ForMathlib.HermitianMat.Rpow

QuantumInfo/ForMathlib/HermitianMat/Rpow.lean on GitHub.

def rpow [source]

Matrix power of a positive semidefinite matrix, as given by the elementwise real power of the diagonal in a diagonalized form.

Note that this has the usual `Real.rpow` caveats, such as 0 to the power -1 giving 0.

```
def rpow (A : HermitianMat d 𝕜) (r : ℝ) : HermitianMat d 𝕜
```

instance instRPow [source]

```
instance instRPow : Pow (HermitianMat d 𝕜) ℝ
```

theorem rpow_conj_unitary [source]

```
theorem rpow_conj_unitary (A : HermitianMat d 𝕜) (U : Matrix.unitaryGroup d 𝕜) (r : ℝ) :
    (HermitianMat.conj U.val A) ^ r = HermitianMat.conj U.val (A ^ r)
```

theorem pow_eq_rpow [source]

```
theorem pow_eq_rpow : A ^ r = A.rpow r
```

theorem rpow_eq_cfc [source]

```
theorem rpow_eq_cfc : A ^ r = A.cfc (· ^ r)
```

theorem diagonal_pow [source]

```
theorem diagonal_pow (f : d → ℝ) :
    (diagonal 𝕜 f) ^ r = diagonal 𝕜 (fun i ↦ (f i) ^ r)
```

theorem rpow_const_continuous [source]

```
@[fun_prop]
theorem rpow_const_continuous {r : ℝ} (hr : 0 ≤ r) : Continuous (fun A : HermitianMat d ℂ ↦ A ^
     r)
```

theorem const_rpow_continuous [source]

```
@[fun_prop]
theorem const_rpow_continuous [NonSingular A] : Continuous (fun r : ℝ ↦ A ^ r)
```

theorem continuousOn_rpow_pos [source]

For a fixed Hermitian matrix A, the function $x \mapsto A^x$ is continuous for $x > 0$.

```
@[fun_prop]
theorem continuousOn_rpow_pos (A : HermitianMat d ℂ) : ContinuousOn (fun x : ℝ ↦ A ^ x) (Set.
    Ioi 0)
```

theorem continuousOn_rpow_neg [source]

For a fixed Hermitian matrix A, the function $x \mapsto A^x$ is continuous for $x < 0$.

```
@[fun_prop]
theorem continuousOn_rpow_neg (A : HermitianMat d ℂ) : ContinuousOn (fun x : ℝ ↦ A ^ x) (Set.
    Iio 0)
```

theorem rpow_one [source]

```
@[simp]
theorem rpow_one : A ^ (1 : ℝ) = A
```

lemma sqrt_eq_cfc_rpow_half [source]

Functional calculus of Real.sqrt is equal to functional calculus of x^(1/2).

```
lemma sqrt_eq_cfc_rpow_half (A : HermitianMat d 𝕜)  :
    A.sqrt = A.cfc (fun x ↦ x ^ (1/2 : ℝ))
```

theorem one_rpow [source]

```
@[simp]
theorem one_rpow : (1 : HermitianMat d 𝕜) ^ r = 1
```

lemma rpow_zero [source]

```
@[simp]
lemma rpow_zero (A : HermitianMat d 𝕜) : A ^ (0 : ℝ) = 1
```

lemma rpow_diagonal [source]

```
lemma rpow_diagonal (a : d → ℝ) (r : ℝ) :
  (diagonal ℂ a) ^ r = diagonal ℂ (fun i => a i ^ r)
```

theorem reindex_rpow [source]

Keeps in line with our simp-normal form for moving reindex outwards.

```
@[simp]
theorem reindex_rpow (e : d ≃ d₂) :
    A.reindex e ^ r = (A ^ r).reindex e
```

theorem mat_rpow_add [source]

```
theorem mat_rpow_add (hA : 0 ≤ A) {p q : ℝ} (hpq : p + q ≠ 0) :
    (A ^ (p + q)).mat = (A ^ p).mat * (A ^ q).mat
```

theorem rpow_mul [source]

```
theorem rpow_mul (hA : 0 ≤ A) {p q : ℝ} : A ^ (p * q) = (A ^ p) ^ q
```

theorem conj_rpow [source]

```
theorem conj_rpow (hA : 0 ≤ A) (hq : q ≠ 0) (hqr : r + 2 * q ≠ 0) :
    (A ^ r).conj (A ^ q) = A ^ (r + 2 * q)
```

theorem pow_half_mul [source]

```
theorem pow_half_mul (hA : 0 ≤ A) :
    (A ^ (1/2 : ℝ)).mat * (A ^ (1/2 : ℝ)).mat = A
```

theorem rpow_pos [source]

```
theorem rpow_pos {A : HermitianMat d 𝕜} (hA : 0 < A) {p : ℝ} : 0 < A ^ p
```

theorem rpow_nonneg [source]

```
theorem rpow_nonneg (hA : 0 ≤ A) {p : ℝ} : 0 ≤ A ^ p
```

theorem inv_eq_rpow_neg_one [source]

```
theorem inv_eq_rpow_neg_one (hA : A.mat.PosDef) : A⁻¹ = A ^ (-1 : ℝ)
```

theorem sandwich_self [source]

```
theorem sandwich_self (hB : B.mat.PosDef) :
    (B.conj (B ^ (-1/2 : ℝ)).mat) = 1
```

lemma rpow_inv_eq_neg_rpow [source]

```
lemma rpow_inv_eq_neg_rpow (hA : A.mat.PosDef) (p : ℝ) : (A ^ p)⁻¹ = A ^ (-p)
```

lemma sandwich_le_one [source]

```
lemma sandwich_le_one (hB : B.mat.PosDef) (h : A ≤ B) :
    (A.conj (B ^ (-1/2 : ℝ)).mat) ≤ 1
```

lemma rpow_neg_mul_rpow_self [source]

```
lemma rpow_neg_mul_rpow_self (hA : A.mat.PosDef) (p : ℝ) :
    (A ^ (-p)).mat * (A ^ p).mat = 1
```

lemma isUnit_rpow_toMat [source]

```
lemma isUnit_rpow_toMat (hA : A.mat.PosDef) (p : ℝ) : IsUnit (A ^ p).mat
```

lemma sandwich_inv [source]

```
lemma sandwich_inv (hB : B.mat.PosDef) :
    (A.conj (B ^ (-1/2 : ℝ)).mat)⁻¹ = A⁻¹.conj (B ^ (1/2 : ℝ)).mat
```

theorem ker_rpow_eq_of_nonneg [source]

```
theorem ker_rpow_eq_of_nonneg {A : HermitianMat d ℂ} (hA : 0 ≤ A) (hp : r ≠ 0):
    (A ^ r).ker = A.ker
```

theorem ker_rpow_le_of_nonneg [source]

```
theorem ker_rpow_le_of_nonneg {A : HermitianMat d ℂ} (hA : 0 ≤ A) :
    (A ^ r).ker ≤ A.ker
```

lemma rpow_kron_diagonal [source]

```
private lemma rpow_kron_diagonal
    (a : d → ℝ) (b : d₂ → ℝ) (r : ℝ) (ha : ∀ i, 0 ≤ a i) (hb : ∀ j, 0 ≤ b j) :
    ((diagonal ℂ a) ⊗ₖ (diagonal ℂ b)) ^ r =
    ((diagonal ℂ a) ^ r) ⊗ₖ ((diagonal ℂ b) ^ r)
```

lemma conj_kron [source]

```
lemma conj_kron
  (A : Matrix d d 𝕜) (B : Matrix d₂ d₂ 𝕜) (C : HermitianMat d 𝕜) (D : HermitianMat d₂ 𝕜) :
    conj (A ⊗ₖ B) (C ⊗ₖ D) = conj A C ⊗ₖ conj B D
```

lemma rpow_kron [source]

```
lemma rpow_kron
    {A : HermitianMat d ℂ} {B : HermitianMat d₂ ℂ} (r : ℝ) (hA : 0 ≤ A) (hB : 0 ≤ B) :
    (A ⊗ₖ B) ^ r = (A ^ r) ⊗ₖ (B ^ r)
```

lemma continuousOn_rpow_uniform [source]

```
lemma continuousOn_rpow_uniform {K : Set ℝ} (hK : IsCompact K) :
    ContinuousOn (fun r : ℝ ↦ UniformOnFun.ofFun {K} (fun t : ℝ ↦ t ^ r)) (Set.Ioi 0)
```

theorem continuousOn_rpow_joint_nonneg_pos [source]

Joint continuity of matrix rpow for Hermitian matrices with positive exponent

```
@[fun_prop]
theorem continuousOn_rpow_joint_nonneg_pos
    {X : Type*} [TopologicalSpace X]
    {A : X → HermitianMat d ℂ} {p : X → ℝ} {S : Set X}
    (hA : ContinuousOn A S) (hp : ContinuousOn p S)
    (hp_pos : ∀ x ∈ S, 0 < p x) :
    ContinuousOn (fun x ↦ A x ^ p x) S
```

theorem cfc_sq_rpow_eq_cfc_rpow [source]

```
theorem cfc_sq_rpow_eq_cfc_rpow
    (A : HermitianMat d ℂ) (hA : 0 ≤ A) (p : ℝ) (hp : 0 < p) :
    (A ^ 2).cfc (fun x => x ^ (p/2)) = A.cfc (fun x => x ^ p)
```

lemma trace_rpow_eq_sum [source]

$Tr[A\hat{}p] = \sum_i \lambda_i\hat{}p$ for a Hermitian matrix A.

```
lemma trace_rpow_eq_sum (A : HermitianMat d ℂ) (p : ℝ) :
    (A ^ p).trace = ∑ i, (A.H.eigenvalues i) ^ p
```

def rpowApprox [source]

Finite integral approximation for the rpow monotonicity proof. Same integrand as `logApprox` but with weight `t ^ q`.

```
noncomputable def rpowApprox (A : HermitianMat d ℂ) (q T : ℝ) : HermitianMat d ℂ
```

theorem rpowApprox_mono [source]

```
theorem rpowApprox_mono {A B : HermitianMat d ℂ} (hA : A.mat.PosDef) (hB : B.mat.PosDef)
    (hAB : A ≤ B) (hq : 0 ≤ q) (T : ℝ) (hT : 0 < T) :
    rpowApprox A q T ≤ rpowApprox B q T
```

def scalarRpowApprox [source]

The scalar function underlying rpowApprox via the CFC.

```
noncomputable def scalarRpowApprox (q T x : ℝ) : ℝ
```

theorem rpowApprox_eq_cfc_scalar [source]

```
theorem rpowApprox_eq_cfc_scalar (A : HermitianMat d ℂ) (hA : A.mat.PosDef) (q T : ℝ)
    (hq : 0 ≤ q) (hT : 0 < T) :
    rpowApprox A q T = A.cfc (scalarRpowApprox q T)
```

def rpowConst [source]

The positive constant arising from the resolvent integral. Equal to `∫ u in Set.Ioi 0, u ^ (q-1) / (1+u) = π / sin(π q)`, but we only need its positivity.

```
noncomputable def rpowConst (q : ℝ) : ℝ
```

lemma rpowConst_integrableOn [source]

The integrand `u ^ (q-1) / (1+u)` is integrable on `(0, ∞)` for `0 < q < 1`.

```
lemma rpowConst_integrableOn (hq : 0 < q) (hq1 : q < 1) :
    IntegrableOn (fun u : ℝ => u ^ (q - 1) / (1 + u)) (Set.Ioi 0)
```

lemma rpowConst_pos [source]

```
lemma rpowConst_pos (hq : 0 < q) (hq1 : q < 1) : 0 < rpowConst q
```

lemma scalarRpowApprox_tendsto [source]

The scalar rpow approximation converges pointwise. `scalarRpowApprox q T x → rpowConst q * (x^q - 1)` as `T → ∞`.

```
lemma scalarRpowApprox_tendsto {x : ℝ} (hx : 0 < x) (hq : 0 < q) (hq1 : q < 1) :
    Filter.Tendsto (fun T => scalarRpowApprox q T x) atTop (nhds (rpowConst q * (x ^ q - 1)))
```

lemma tendsto_rpowApprox [source]

The matrix rpow approximation converges: `rpowApprox A q T → rpowConst q • (A^q - 1)`.

```
lemma tendsto_rpowApprox (hA : A.mat.PosDef) (hq : 0 < q) (hq1 : q < 1) :
    Tendsto (rpowApprox A q) atTop (nhds (rpowConst q • (A ^ q - 1)))
```

theorem rpow_le_rpow_of_posDef [source]

```
theorem rpow_le_rpow_of_posDef (hA : A.mat.PosDef) (hAB : A ≤ B)
    (hq : 0 < q) (hq1 : q ≤ 1) : A ^ q ≤ B ^ q
```

theorem rpow_le_rpow_of_le [source]

*The **Löwner—Heinz theorem**: for matrices A and B, if `0 ≤ A ≤ B` and `0 < q ≤ 1`, then `A^q ≤ B^q`. That is, real roots are operator monotone.*

```
theorem rpow_le_rpow_of_le (hA : 0 ≤ A) (hAB : A ≤ B)
    (hq : 0 < q) (hq1 : q ≤ 1) : A ^ q ≤ B ^ q
```

lemma lieb_thirring_le_one [source]

An inequality of Lieb-Thirring type. For $0 < r \leq 1$: `Tr[B^r A^r B^r] ≤ Tr[(B A B)^r]`.

```
lemma lieb_thirring_le_one
    {A B : HermitianMat d ℂ} (hA : 0 ≤ A) (hB : 0 ≤ B)
    {r : ℝ} (hq0 : 0 < r) (hq1 : q ≤ r) :
    ((A ^ r).conj (B ^ r).mat).trace ≤ ((A.conj B.mat) ^ r).trace
```

lemma trace_rpow_add_le [source]

```
lemma trace_rpow_add_le
    {A B : HermitianMat d ℂ} (hA : 0 ≤ A) (hB : 0 ≤ B)
    (p : ℝ) (hp : 0 < p) (hp1 : p ≤ 1) :
    ((A + B) ^ p).trace ≤ (A ^ p).trace + (B ^ p).trace
```

2.65 QuantumInfo.ForMathlib.HermitianMat.Schatten

QuantumInfo/ForMathlib/HermitianMat/Schatten.lean on GitHub.

def schattenNorm [source]

*The Schatten p-norm of a matrix A is (Tr[(A*A)^(p/2)])^(1/p).*

```
noncomputable def schattenNorm (A : Matrix d d ℂ) (p : ℝ) : ℝ
```

theorem schattenNorm_hermitian_pow [source]

```
theorem schattenNorm_hermitian_pow {A : HermitianMat d ℂ} (hA : 0 ≤ A) {p : ℝ} (hp : 0 < p) :
    schattenNorm A.mat p = (A ^ p).trace ^ (1/p)
```

lemma schattenNorm_nonneg [source]

```
lemma schattenNorm_nonneg (A : Matrix d d ℂ) (p : ℝ) :
    0 ≤ schattenNorm A p
```

lemma schattenNorm_pow_eq [source]

```
lemma schattenNorm_pow_eq
  (A : HermitianMat d ℂ) (hA : 0 ≤ A) (p k : ℝ) (hp : 0 < p) (hk : 0 < k) :
    schattenNorm (A ^ k).mat p = (schattenNorm A.mat (k * p)) ^ k
```

lemma trace_eq_schattenNorm_rpow [source]

```
lemma trace_eq_schattenNorm_rpow
    (A : HermitianMat d ℂ) (hA : 0 ≤ A) (r : ℝ) (hr : 0 < r) :
    (A ^ r).trace = (schattenNorm A.mat r) ^ r
```

lemma schattenNorm_trace_as_eigenvalue_sum [source]

```
lemma schattenNorm_trace_as_eigenvalue_sum (A : Matrix d d ℂ) (p : ℝ) :
    RCLike.re ((Matrix.isHermitian_mul_conjTranspose_self A.conjTranspose).cfc (· ^ (p/2))).
    trace =
    ∑ i : d, ((Matrix.isHermitian_mul_conjTranspose_self A.conjTranspose).eigenvalues i) ^ (p
    /2)
```

lemma schattenNorm_rpow_eq_sum_singularValues [source]

```
lemma schattenNorm_rpow_eq_sum_singularValues (A : Matrix d d ℂ) {p : ℝ} (hp : 0 < p) :
    schattenNorm A p ^ p = ∑ i : d, singularValues A i ^ p
```

lemma schattenNorm_eq_sum_singularValues_rpow [source]

```
lemma schattenNorm_eq_sum_singularValues_rpow (A : Matrix d d ℂ) {p : ℝ} (hp : 0 < p) :
    schattenNorm A p = (∑ i : d, singularValues A i ^ p) ^ (1/p)
```

lemma schattenNorm_rpow_eq_sum_sorted [source]

||A||_p^p equals the same sum over sorted singular values.

```
lemma schattenNorm_rpow_eq_sum_sorted (A : Matrix d d ℂ) {p : ℝ} (hp : 0 < p) :
    schattenNorm A p ^ p =
    ∑ i : Fin (Fintype.card d), singularValuesSorted A i ^ p
```

lemma trace_young [source]

Scalar trace Young inequality for PSD matrices: ⟨⟨A, B⟩⟩ ≤ Tr[A^p]/p + Tr[B^q]/q for PSD A, B and conjugate p, q > 1.

```
lemma HermitianMat.trace_young
    (A B : HermitianMat d ℂ) (hA : 0 ≤ A) (hB : 0 ≤ B)
    (p q : ℝ) (hp : 1 < p) (hpq : 1/p + 1/q = 1) :
    ⟪A, B⟫_ℝ ≤ (A ^ p).trace / p + (B ^ q).trace / q
```

lemma conjTranspose_half_mul_eq_conj [source]

For PSD `A` and Hermitian `B`, the product `C = A^{1/2} * B` satisfies `C^* C = (A.conj B.mat).mat = B * A * B`.

```
lemma conjTranspose_half_mul_eq_conj
    {A B : HermitianMat d ℂ} (hA : 0 ≤ A) :
    ((A ^ (1/2 : ℝ)).mat * B.mat).conjTranspose * ((A ^ (1/2 : ℝ)).mat * B.mat)
    = (A.conj B.mat).mat
```

lemma schattenNorm_half_mul_rpow_eq_trace_conj [source]

```
lemma schattenNorm_half_mul_rpow_eq_trace_conj
    {A B : HermitianMat d ℂ} (hA : 0 ≤ A)
    {α : ℝ} (hα : 0 < α) :
    (schattenNorm ((A ^ (1/2 : ℝ)).mat * B.mat) (2 * α)) ^ (2 * α) =
    ((A.conj B.mat) ^ α).trace
```

lemma schattenNorm_mul_le [source]

```
lemma schattenNorm_mul_le (A B : Matrix d d ℂ) {r p q : ℝ}
    (hr : 0 < r) (hp : 0 < p) (hq : 0 < q) (hpqr : 1 / r = 1 / p + 1 / q) :
    schattenNorm (A * B) r ≤ schattenNorm A p * schattenNorm B q
```

lemma trace_rpow_conj_le [source]

```
lemma HermitianMat.trace_rpow_conj_le
    {A B : HermitianMat d ℂ} (hA : 0 ≤ A) (hB : 0 ≤ B)
    {α p q : ℝ} (hα : 0 < α) (hp : 0 < p) (hq : 0 < q)
    (hpq : 1 / (2 * α) = 1 / p + 1 / q) :
    ((A.conj B.mat) ^ α).trace ≤
    (((A ^ (p / 2)).trace) ^ (1 / p) * ((B ^ q).trace) ^ (1 / q)) ^ (2 * α)
```

2.66 QuantumInfo.ForMathlib.HermitianMat.Sqrt

`QuantumInfo/ForMathlib/HermitianMat/Sqrt.lean` on GitHub.

def sqrt [source]

The square root of a Hermitian matrix. Negative eigenvalues are mapped to zero.

```
noncomputable def sqrt (A : HermitianMat d 𝕜) : HermitianMat d 𝕜
```

theorem sqrt_sq_eq_proj [source]

```
theorem sqrt_sq_eq_proj (A : HermitianMat d 𝕜) :
    A.sqrt.mat * A.sqrt.mat = A⁺
```

theorem sqrt_sq [source]

```
theorem sqrt_sq (hA : 0 ≤ A) :
    A.sqrt.mat * A.sqrt.mat = A
```

theorem commute_sqrt_left [source]

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem commute_sqrt_left (hAB : Commute A.mat B.mat) :
    Commute A.sqrt.mat B.mat
```

theorem commute_sqrt_right [source]

```
@[aesop unsafe apply 50% (rule_sets := [Commutes])]
theorem commute_sqrt_right (hAB : Commute A.mat B.mat) :
    Commute A.mat B.sqrt.mat
```

lemma sqrt_inv_mul_self_mul_sqrt_inv_eq_one [source]

*For a positive definite matrix A, A^{-1/2} * A * A^{-1/2} = I.*

```
lemma sqrt_inv_mul_self_mul_sqrt_inv_eq_one {A : HermitianMat d 𝕜} (hA : A.mat.PosDef) :
    A⁻¹.sqrt.mat * A.mat * A⁻¹.sqrt.mat = 1
```

theorem sqrt_nonneg [source]

```
theorem sqrt_nonneg (A : HermitianMat d 𝕜) : 0 ≤ A.sqrt
```

theorem sqrt_pos [source]

```
theorem sqrt_pos (h : 0 < A) : 0 < A.sqrt
```

theorem sqrt_posDef [source]

```
theorem sqrt_posDef {A : HermitianMat d 𝕜} (hA : A.mat.PosDef) :
    A.sqrt.mat.PosDef
```

def evalHermitianMatSqrt [source]

Positivity extension for `HermitianMat.sqrt`

```
@[positivity HermitianMat.sqrt _]
def evalHermitianMatSqrt : PositivityExt where eval {_u _α} _zα _pα e
```

2.67 QuantumInfo.ForMathlib.HermitianMat.Trace

QuantumInfo/ForMathlib/HermitianMat/Trace.lean on GitHub.

def trace [source]

The trace of the matrix. This requires a `IsMaximalSelfAdjoint R α` instance, and then maps from `HermitianMat n α` to `R`. This means that the trace of (say) a `HermitianMat n ℤ` gives values in ℤ, but that the trace of a `HermitianMat n ℂ` gives values in ℝ. The fact that traces are "automatically" real reduces coercions down the line.

```
def trace (A : HermitianMat n α) : R
```

theorem trace_eq_trace [source]

`HermitianMat.trace` reduces to `Matrix.trace` in the algebra.

```
theorem trace_eq_trace (A : HermitianMat n α) : algebraMap R α A.trace = Matrix.trace A.mat
```

theorem trace_smul [source]

```
@[simp]
theorem trace_smul (A : HermitianMat n α) (r : R) : (r • A).trace = r * A.trace
```

theorem trace_zero [source]

```
@[simp]
theorem trace_zero : (0 : HermitianMat n α).trace = 0
```

theorem trace_add [source]

```
@[simp]
theorem trace_add (A B : HermitianMat n α) : (A + B).trace = A.trace + B.trace
```

theorem trace_neg [source]

```
@[simp]
theorem trace_neg (A : HermitianMat n α) : (-A).trace = -A.trace
```

theorem trace_sub [source]

```
@[simp]
theorem trace_sub (A B : HermitianMat n α) : (A - B).trace = A.trace - B.trace
```

theorem isSelfAdjoint_trace [source]

```
theorem _root_.Matrix.IsHermitian.isSelfAdjoint_trace {A : Matrix n n α} (hA : A.IsHermitian) :
    IsSelfAdjoint A.trace
```

theorem trace_kronecker [source]

```
@[simp]
theorem trace_kronecker [FaithfulSMul R α] : (A ⊗ₖ B).trace = A.trace * B.trace
```

theorem trace_eq_trace_trivial [source]

`HermitianMat.trace` reduces to `Matrix.trace` when the elements are a `TrivialStar`.

```
@[simp]
theorem trace_eq_trace_trivial (A : HermitianMat n ℝ) : A.trace = A.mat.trace
```

theorem trace_eq_re_trace [source]

```
theorem trace_eq_re_trace (A : HermitianMat n 𝕜) : A.trace = RCLike.re A.mat.trace
```

theorem trace_one [source]

```
@[simp]
theorem trace_one [DecidableEq n] : (1 : HermitianMat n 𝕜).trace = Fintype.card n
```

theorem trace_eq_trace_rc [source]

`HermitianMat.trace` reduces to `Matrix.trace` when the elements are `RCLike`.

```
@[simp]
theorem trace_eq_trace_rc (A : HermitianMat n 𝕜) : A.trace = A.mat.trace
```

theorem trace_diagonal [source]

```
theorem trace_diagonal {T : Type*} [Fintype T] [DecidableEq T] (f : T → ℝ) :
    (diagonal 𝕜 f).trace = ∑ i, f i
```

theorem sum_eigenvalues_eq_trace [source]

```
theorem sum_eigenvalues_eq_trace [DecidableEq n] (A : HermitianMat n 𝕜) :
    ∑ i, A.H.eigenvalues i = A.trace
```

theorem trace_eq_zero_iff [source]

```
theorem trace_eq_zero_iff (A : HermitianMat n 𝕜) : A.trace = 0 ↔ A.mat.trace = 0
```

theorem trace_eq_one_iff [source]

```
theorem trace_eq_one_iff (A : HermitianMat n 𝕜) : A.trace = 1 ↔ A.mat.trace = 1
```

theorem trace_reindex [source]

```
@[simp]
theorem trace_reindex (A : HermitianMat n ℂ) (e : n ≃ m) :
    (A.reindex e).trace = A.trace
```

def traceLeft [source]

```
def traceLeft (A : HermitianMat (m × n) α) : HermitianMat n α
```

def traceRight [source]

```
def traceRight (A : HermitianMat (m × n) α) : HermitianMat m α
```

theorem traceLeft_mat [source]

```
@[simp]
theorem traceLeft_mat : A.traceLeft.mat = A.mat.traceLeft
```

theorem traceLeft_add [source]

```
@[simp]
theorem traceLeft_add : (A + B).traceLeft = A.traceLeft + B.traceLeft
```

theorem traceLeft_neg [source]

```
@[simp]
theorem traceLeft_neg : (-A).traceLeft = -A.traceLeft
```

theorem traceLeft_sub [source]

```
@[simp]
theorem traceLeft_sub : (A - B).traceLeft = A.traceLeft - B.traceLeft
```

theorem traceRight_mat [source]

```
@[simp]
theorem traceRight_mat :
    (traceRight A).mat = A.mat.traceRight
```

theorem traceRight_add [source]

```
@[simp]
theorem traceRight_add : (A + B).traceRight = A.traceRight + B.traceRight
```

theorem traceRight_neg [source]

```
@[simp]
theorem traceRight_neg : (-A).traceRight = -A.traceRight
```

theorem traceRight_sub [source]

```
@[simp]
theorem traceRight_sub : (A - B).traceRight = A.traceRight - B.traceRight
```

theorem traceLeft_smul [source]

```
@[simp]
theorem traceLeft_smul (r : ℝ) : (r • A).traceLeft = r • A.traceLeft
```

theorem traceRight_smul [source]

```
@[simp]
theorem traceRight_smul (r : ℝ) : (r • A).traceRight = r • A.traceRight
```

theorem traceLeft_trace [source]

```
@[simp]
theorem traceLeft_trace : A.traceLeft.trace = A.trace
```

theorem traceRight_trace [source]

```
@[simp]
theorem traceRight_trace : A.traceRight.trace = A.trace
```

theorem traceLeft_kron [source]

```
@[simp]
theorem traceLeft_kron [Fintype m] : (A ⊗ₖ B).traceLeft = A.trace • B
```

theorem traceRight_kron [source]

```
@[simp]
theorem traceRight_kron [Fintype n] : (A ⊗ₖ B).traceRight = B.trace • A
```

2.68 QuantumInfo.ForMathlib.HermitianMat.Unitary

QuantumInfo/ForMathlib/HermitianMat/Unitary.lean on GitHub.

theorem neg_unitary_val [source]

```
@[simp]
theorem neg_unitary_val (u : 𝐔[α]) : (-u).val = -u
```

theorem star_kron [source]

```
@[simp]
theorem star_kron (a : Matrix α α ℂ) (b : Matrix β β ℂ) : star (a ⊗ₖ b) = (star a) ⊗ₖ (star b)
```

theorem kron_unitary [source]

```
theorem kron_unitary (a : 𝐔[α]) (b : 𝐔[β]) : a.val ⊗ₖ b.val ∈ 𝐔[α × β]
```

def unitary_kron [source]

```
def unitary_kron (a : 𝐔[α]) (b : 𝐔[β]) : 𝐔[α × β]
```

theorem unitary_kron_apply [source]

```
@[simp]
theorem unitary_kron_apply (a : 𝐔[α]) (b : 𝐔[β]) (i₁ i₂ : α) (j₁ j₂ : β) :
    (a ⊗ᵤ b) (i₁, j₁) (i₂, j₂) = (a i₁ i₂) * (b j₁ j₂)
```

theorem unitary_kron_one_one [source]

```
@[simp]
theorem unitary_kron_one_one : (1 : 𝐔[α]) ⊗ᵤ (1 : 𝐔[β]) = (1 : 𝐔[α × β])
```

lemma unitary_row_sum_norm_sq [source]

For a unitary matrix C, the row sums of ‖C i j‖^2 equal 1.

```
lemma unitary_row_sum_norm_sq (C : Matrix d d ℂ) (hC : C * C.conjTranspose = 1) (i : d) :
    ∑ j, ‖C i j‖ ^ 2 = 1
```

lemma unitary_col_sum_norm_sq [source]

For a unitary matrix C, the column sums of ‖C i j‖^2 equal 1.

```
lemma unitary_col_sum_norm_sq (C : Matrix d d ℂ) (hC : C.conjTranspose * C = 1) (j : d) :
    ∑ i, ‖C i j‖ ^ 2 = 1
```

2.69 QuantumInfo.ForMathlib.IsMaximalSelfAdjoint

QuantumInfo/ForMathlib/IsMaximalSelfAdjoint.lean on GitHub.

class IsMaximalSelfAdjoint [source]

```
class IsMaximalSelfAdjoint (R : outParam Type*) (α : Type*) [Star α] [Star R] [CommSemiring R]
    [Semiring α] [TrivialStar R] [Algebra R α] where
  selfadjMap : α →+ R
  selfadj_smul : ∀ (r : R) (a : α), selfadjMap (r • a) = r * (selfadjMap a)
  selfadj_algebra : ∀ {a : α}, IsSelfAdjoint a → algebraMap _ _ (selfadjMap a) = a
```

instance instTrivialStar_IsMaximalSelfAdjoint [source]

Every `TrivialStar CommSemiring` is its own maximal self adjoints.

```
instance instTrivialStar_IsMaximalSelfAdjoint {R} [Star R] [TrivialStar R] [CommSemiring R] :
    IsMaximalSelfAdjoint R R where
```

instance instRCLike_IsMaximalSelfAdjoint [source]

ℝ is the maximal self adjoint elements over RCLike

```
instance instRCLike_IsMaximalSelfAdjoint {α} [RCLike α] : IsMaximalSelfAdjoint ℝ α where
```

theorem trivial_selfadjMap [source]

```
@[simp]
theorem trivial_selfadjMap {R} [Star R] [TrivialStar R] [CommSemiring R] : (selfadjMap : R →+ R
    ) = .id R
```

theorem RCLike_selfadjMap [source]

```
@[simp]
theorem RCLike_selfadjMap {α} [RCLike α] : (selfadjMap : α →+ ℝ) = RCLike.re
```

2.70 QuantumInfo.ForMathlib.Isometry

QuantumInfo/ForMathlib/Isometry.lean on GitHub.

def Isometry [source]

An isometry is a matrix `A` such that AA^H `= 1`. Compare with a unitary, which requires AA^H `=` $A^H A$ `= 1`. It is common to claim that, in a finite-dimensional vector space, a two-sided isometry (`A.Isometry ∧` A^H`.Isometry`) must be square and therefore unitary; this is does not work out so well here, since a `Matrix m n R` can be a two-sided isometry, but cannot be a `unitary` since the rows and columns are index by different labels.

```
def Matrix.Isometry (A : Matrix d d₂ R) : Prop
```

theorem submatrix_one_isometry [source]

```
theorem Matrix.submatrix_one_isometry {e : d₂ → d} {f : d₃ → d} (he : e.Bijective) (hf : f.
    Injective) :
    (submatrix (α := R) 1 e f).Isometry
```

theorem submatrix_one_id_left_isometry [source]

```
theorem Matrix.submatrix_one_id_left_isometry {e : d₂ → d} (he : e.Bijective) :
    (submatrix (1 : Matrix d d R) e id).Isometry
```

theorem submatrix_one_id_right_isometry [source]

```
theorem Matrix.submatrix_one_id_right_isometry {e : d₂ → d} (he : e.Injective) :
    (submatrix (1 : Matrix d d R) id e).Isometry
```

theorem mem_unitaryGroup_iff_isometry [source]

```
theorem Matrix.mem_unitaryGroup_iff_isometry (A : Matrix d d R) :
    A ∈ unitaryGroup d R ↔ A.Isometry ∧ Aᴴ.Isometry
```

theorem permMatrix_mem_unitaryGroup [source]

```
theorem Equiv.Perm.permMatrix_mem_unitaryGroup (e : Perm d) :
    e.permMatrix R ∈ Matrix.unitaryGroup d R
```

theorem reindex_one_isometry [source]

```
theorem Matrix.reindex_one_isometry (e : d ≃ d₂) (f : d ≃ d₃) :
    (reindex (α := R) e f 1).Isometry
```

theorem reindex_one_mem_unitaryGroup [source]

```
theorem Matrix.reindex_one_mem_unitaryGroup (e : d ≃ d₂)  :
    reindex (α := R) e e 1 ∈ unitaryGroup d₂ R
```

theorem reindex_eq_conj [source]

```
theorem Matrix.reindex_eq_conj (A : Matrix d d R) (e : d ≃ d₂) : reindex e e A =
    (reindex (α := R) e (.refl d) 1) * A * (reindex (α := R) (.refl d) e 1)
```

theorem reindex_eq_conj_unitaryGroup' [source]

```
theorem Matrix.reindex_eq_conj_unitaryGroup' (A : Matrix d d R) (e : Equiv.Perm d) : reindex e
    e A =
    (⟨_, e⁻¹.permMatrix_mem_unitaryGroup⟩ : unitaryGroup d R) * A * (⟨_, e.
    permMatrix_mem_unitaryGroup⟩ : unitaryGroup d R)
```

theorem eigenvalue_ext [source]

```
theorem Matrix.IsHermitian.eigenvalue_ext (hA : A.IsHermitian)
  (h : ∀ (v : d → 𝕜) (lam : 𝕜), A *ᵥ v = lam • v → B *ᵥ v = lam • v) :
    A = B
```

theorem cfc_eq_any_isometry [source]

Generalizes `Matrix.IsHermitian.cfc.eq_1`, which gives a definition for the matrix CFC in terms of `Matrix.IsHermitian.eigenvalues` and `Matrix.IsHermitian.eigenvectorUnitary`, to show that the CFC works similarly for _any_ diagonalization by a two-sided isometry.

```
theorem Matrix.IsHermitian.cfc_eq_any_isometry {n m 𝕜 : Type*} [RCLike 𝕜]
  [Fintype n] [DecidableEq n] [Fintype m] [DecidableEq m]
  {A : Matrix n n 𝕜} (hA : A.IsHermitian) {U : Matrix n m 𝕜}
  (hU₁ : U * Uᴴ = 1) (hU₂ : Uᴴ * U = 1) {D : m → ℝ}
  (hUD : A = (U * diagonal (RCLike.ofReal ∘ D) : Matrix _ _ _) * Uᴴ) (f : ℝ → ℝ) :
    hA.cfc f = (U * diagonal (RCLike.ofReal ∘ f ∘ D) : Matrix _ _ _) * Uᴴ
```

theorem cfc_eq_any_unitary [source]

Generalizes `Matrix.IsHermitian.cfc.eq_1`, which gives a definition for the matrix CFC in terms of `Matrix.IsHermitian.eigenvalues` and `Matrix.IsHermitian.eigenvectorUnitary`, to show that the CFC works similarly for _any_ diagonalization.

```
theorem Matrix.IsHermitian.cfc_eq_any_unitary {n 𝕜 : Type*} [RCLike 𝕜] [Fintype n] [DecidableEq
    n]
  {A : Matrix n n 𝕜} (hA : A.IsHermitian) {U : unitaryGroup n 𝕜} {D : n → ℝ}
  (hUD : A = U.val * diagonal (RCLike.ofReal ∘ D) * star U.val) (f : ℝ → ℝ) :
    hA.cfc f = U.val * diagonal (RCLike.ofReal ∘ f ∘ D) * star U.val
```

theorem cfc_conj_isometry' [source]

```
private theorem Matrix.cfc_conj_isometry' (hA : A.IsHermitian) (f : ℝ → ℝ) {u : Matrix d₂ d 𝕜}
  (hu₁ : u.Isometry) (hu₂ : uᴴ.Isometry) :
    cfc f (u * A * uᴴ) = u * (cfc f A) * uᴴ
```

theorem cfc_conj_isometry [source]

```
theorem Matrix.cfc_conj_isometry (f : ℝ → ℝ) {u : Matrix d₂ d 𝕜}
  (hu₁ : u.Isometry) (hu₂ : uᴴ.Isometry) :
    cfc f (u * A * uᴴ) = u * (cfc f A) * uᴴ
```

theorem cfc_conj_unitary [source]

```
theorem Matrix.cfc_conj_unitary (f : ℝ → ℝ) (u : unitaryGroup d 𝕜) :
    cfc f (u * A * u⁻¹) = u * (cfc f A) * u⁻¹
```

theorem cfc_conj_unitary' [source]

```
theorem Matrix.cfc_conj_unitary' (f : ℝ → ℝ) (u : unitaryGroup d 𝕜) :
    cfc f (u^H * A * u.val) = u^H * (cfc f A) * u.val
```

theorem cfc_reindex [source]

```
theorem Matrix.cfc_reindex (f : ℝ → ℝ) (e : d ≃ d₂) :
    cfc f (reindex e e A) = reindex e e (cfc f A)
```

theorem commute_euclideanLin [source]

```
theorem Matrix.commute_euclideanLin (hAB : Commute A B) :
    Commute A.toEuclideanLin B.toEuclideanLin
```

theorem orthogonalFamily_eigenspace_inf_eigenspace' [source]

Similar to `LinearMap.IsSymmetric.orthogonalFamily_eigenspace_inf_eigenspace`, but here the direct sum is indexed by only the pairs of eigenvalues, as opposed to all pairs of 𝕜 values, giving a finite decomposition.

```
theorem LinearMap.IsSymmetric.orthogonalFamily_eigenspace_inf_eigenspace' {𝕜 E : Type*} [RCLike
    𝕜]
  [NormedAddCommGroup E] [InnerProductSpace 𝕜 E] {A B : E →ₗ[𝕜] E}
  (hA : A.IsSymmetric) (hB : B.IsSymmetric) :
    OrthogonalFamily 𝕜 (fun (μ₁₂ : Eigenvalues A × Eigenvalues B) ↦
      ↥(eigenspace A μ₁₂.1 ⊓ eigenspace B μ₁₂.2)) fun μ₁₂ ↦
    (eigenspace A μ₁₂.1 ⊓ eigenspace B μ₁₂.2).subtypeₗᵢ
```

theorem iSup_mono_bot [source]

Variant of `iSup_mono'` that allows for an easier handling of bottom elements.

```
theorem iSup_mono_bot {α : Type*} {ι ι' : Sort*} [CompleteLattice α]
  {f : ι → α} {g : ι' → α} (h : ∀ (i : ι), f i = ⊥ ∨ ∃ i', f i ≤ g i') :
    iSup f ≤ iSup g
```

def isSymmetric_directSumDecomposition [source]

```
noncomputable def Commute.isSymmetric_directSumDecomposition  {𝕜 E : Type*} [RCLike 𝕜]
  [NormedAddCommGroup E] [InnerProductSpace 𝕜 E] {A B : E →ₗ[𝕜] E} [FiniteDimensional 𝕜 E]
  (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) :
    DirectSum.Decomposition fun (μ₁₂ : Eigenvalues A × Eigenvalues B) ↦
      (eigenspace A μ₁₂.1 ⊓ eigenspace B μ₁₂.2)
```

theorem directSum_isInternal_of_commute' [source]

Similar to `LinearMap.IsSymmetric.directSum_isInternal_of_commute`, but here the direct sum is indexed by only the pairs of eigenvalues, as opposed to all pairs of $\mathbb{k}$ values, giving a finite decomposition.

```
theorem LinearMap.IsSymmetric.directSum_isInternal_of_commute' {𝕜 E : Type*} [RCLike 𝕜]
  [NormedAddCommGroup E] [InnerProductSpace 𝕜 E] {A B : E →ₗ[𝕜] E} [FiniteDimensional 𝕜 E]
  (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) :
    DirectSum.IsInternal fun (μ₁₂ : Eigenvalues A × Eigenvalues B) ↦
      eigenspace A μ₁₂.1 ⊓ eigenspace B μ₁₂.2
```

def LinearMap.sharedEigenbasis [source]

```
noncomputable def LinearMap.sharedEigenbasis {A B : EuclideanSpace 𝕜 d →ₗ[𝕜] EuclideanSpace 𝕜 d
    }
  (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) :
    OrthonormalBasis d 𝕜 (EuclideanSpace 𝕜 d)
```

def sharedEigenvaluesA [source]

```
noncomputable def LinearMap.sharedEigenvaluesA {A B : EuclideanSpace 𝕜 d →ₗ[𝕜] EuclideanSpace
    𝕜 d}
    (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) : d → ℝ
```

def sharedEigenvaluesB [source]

```
noncomputable def LinearMap.sharedEigenvaluesB {A B : EuclideanSpace 𝕜 d →ₗ[𝕜] EuclideanSpace
    𝕜 d}
    (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) : d → ℝ
```

theorem mem_eigenspace_inf_of_sharedEigenbasis [source]

```
theorem LinearMap.mem_eigenspace_inf_of_sharedEigenbasis {A B : EuclideanSpace 𝕜 d →ₗ[𝕜]
    EuclideanSpace 𝕜 d}
    (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) (i : d) :
    ∃ (μ : Module.End.Eigenvalues A) (ν : Module.End.Eigenvalues B),
      LinearMap.sharedEigenbasis hA hB hAB i ∈ Module.End.eigenspace A μ ⊓ Module.End.
    eigenspace B ν
```

theorem apply_A_sharedEigenbasis [source]

```
theorem LinearMap.apply_A_sharedEigenbasis {A B : EuclideanSpace 𝕜 d →ₗ[𝕜] EuclideanSpace 𝕜 d}
    (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) (i : d) :
    A (sharedEigenbasis hA hB hAB i) = (sharedEigenvaluesA hA hB hAB i : 𝕜) • (sharedEigenbasis
     hA hB hAB i)
```

theorem apply_B_sharedEigenbasis [source]

```
theorem LinearMap.apply_B_sharedEigenbasis {A B : EuclideanSpace 𝕜 d →ₗ[𝕜] EuclideanSpace 𝕜 d}
    (hA : A.IsSymmetric) (hB : B.IsSymmetric) (hAB : Commute A B) (i : d) :
    B (sharedEigenbasis hA hB hAB i) = (sharedEigenvaluesB hA hB hAB i : 𝕜) • (sharedEigenbasis
     hA hB hAB i)
```

def Matrix.sharedEigenbasis [source]

```
noncomputable def Matrix.sharedEigenbasis
  (hA : A.IsHermitian) (hB : B.IsHermitian) (hAB : Commute A B) :
    OrthonormalBasis d 𝕜 (EuclideanSpace 𝕜 d)
```

def sharedEigenvectorUnitary [source]

```
noncomputable def Matrix.sharedEigenvectorUnitary (hA : A.IsHermitian) (hB : B.IsHermitian)
    (hAB : Commute A B) : Matrix.unitaryGroup d 𝕜
```

theorem sharedEigenvectorUnitary_mulVec [source]

Analogous to `Matrix.IsHermitian.eigenvectorUnitary_mulVec` for the shared basis.

```
theorem sharedEigenvectorUnitary_mulVec (j : d) : (sharedEigenvectorUnitary hA hB hAB) *ᵥ
    Pi.single j 1 = WithLp.ofLp (sharedEigenbasis hA hB hAB j)
```

def sharedEigenvalueA [source]

```
noncomputable def sharedEigenvalueA (j : d) : ℝ
```

def sharedEigenvalueB [source]

```
noncomputable def sharedEigenvalueB (j : d) : ℝ
```

theorem mulVec_sharedEigenbasisA [source]

Analogous to `Matrix.IsHermitian.mulVec_eigenvectorBasis` for the shared basis.

```
theorem mulVec_sharedEigenbasisA (j : d) :
    A *ᵥ (sharedEigenbasis hA hB hAB j) =
    (sharedEigenvalueA hA hB hAB) j • WithLp.ofLp (sharedEigenbasis hA hB hAB j)
```

theorem mulVec_sharedEigenbasisB [source]

```
theorem mulVec_sharedEigenbasisB (j : d) :
    B *ᵥ (sharedEigenbasis hA hB hAB j) =
    (sharedEigenvalueB hA hB hAB) j • WithLp.ofLp (sharedEigenbasis hA hB hAB j)
```

theorem star_shared_mul_A_mul_IsDiag [source]

```
theorem star_shared_mul_A_mul_IsDiag : IsDiag
    ((star (sharedEigenvectorUnitary hA hB hAB : Matrix d d 𝕜)) * A *
      (sharedEigenvectorUnitary hA hB hAB : Matrix d d 𝕜))
```

theorem star_shared_mul_B_mul_IsDiag [source]

Analogous to `Matrix.IsHermitian.star_mul_self_mul_eq_diagonal` for the shared basis.

```
theorem star_shared_mul_B_mul_IsDiag : IsDiag
    ((star (sharedEigenvectorUnitary hA hB hAB : Matrix d d 𝕜)) * B *
```

```
    (sharedEigenvectorUnitary hA hB hAB : Matrix d d 𝕜))
```

theorem exists_unitary [source]

```
theorem Commute.exists_unitary (hA : A.IsHermitian) (hB : B.IsHermitian) (hAB : Commute A B) :
    ∃ U : Matrix.unitaryGroup d 𝕜, (U.val * A * U^H).IsDiag ∧ (U.val * B * U^H).IsDiag
```

instance instInvertibleUnitaryGroup [source]

```
instance instInvertibleUnitaryGroup (U : Matrix.unitaryGroup d 𝕜) : Invertible U
```

instance Invertible U.val [source]

```
instance (U : Matrix.unitaryGroup d 𝕜) : Invertible U.val
```

theorem IsDiag.exists_cfc [source]

If a matrix is diagonalized by a unitary matrix, then it can be written as a CFC of a (particular, canonical) diagonal matrix.

```
theorem Matrix.IsDiag.exists_cfc {U : Matrix.unitaryGroup d 𝕜} {M : Matrix d d 𝕜}
  (hU : (U.val * M * U^H).IsDiag) (hM : M.IsHermitian) (e : d ≃ Fin (Fintype.card d)) :
      ∃ f : ℝ → ℝ,
    M = cfc f (U^H * (Matrix.diagonal fun x => ↑↑(e x)) * U.val)
```

theorem Commute.exists_cfc [source]

If two Hermitian matrices commute, there exists a common matrix that they are both a CFC of.

```
theorem Commute.exists_cfc (hA : A.IsHermitian) (hB : B.IsHermitian) (hAB : Commute A B) :
    ∃ C : Matrix d d 𝕜, (∃ f : ℝ → ℝ, A = cfc f C) ∧ (∃ g : ℝ → ℝ, B = cfc g C)
```

2.71 QuantumInfo.ForMathlib.Lieb

QuantumInfo/ForMathlib/Lieb.lean on GitHub.

theorem LiebConcavity [source]

```
theorem LiebConcavity (K : Matrix n m ℂ) (hq : 0 ≤ q) (hr : 0 ≤ r) (hqr : q + r ≤ 1) :
  let F : (HermitianMat m ℂ × HermitianMat n ℂ) → ℝ
```

2.72 QuantumInfo.ForMathlib.LimSupInf

QuantumInfo/ForMathlib/LimSupInf.lean on GitHub.

lemma exists_strictMono_seq_le [source]

```
lemma exists_strictMono_seq_le (y : ℝ≥0) (f : ℝ≥0 → ℕ → ℝ≥0∞) (hf : ∀ x > 0, Filter.atTop.
    liminf (f x) ≤ y) :
    ∃ n : ℕ → ℕ, StrictMono n ∧ ∀ k : ℕ, f ((k : ℝ≥0) + 1)⁻¹ (n k) ≤ (y : ℝ≥0∞) + ((k : ℝ≥0) +
    1)⁻¹
```

lemma exists_seq_bound [source]

```
lemma exists_seq_bound (y : ℝ≥0) (f : ℝ≥0 → ℕ → ℝ≥0∞) (hf : ∀ x > 0, Filter.atTop.limsup (f x)
    ≤ y) :
    ∃ M : ℕ → ℕ, StrictMono M ∧ ∀ k, ∀ n ≥ M k, f ((k + 1 : ℝ≥0)⁻¹) n ≤ y + (k + 1 : ℝ≥0∞)⁻¹
```

lemma exists_liminf_zero_of_forall_liminf_le [source]

```
lemma exists_liminf_zero_of_forall_liminf_le (y : ℝ≥0) (f : ℝ≥0 → ℕ → ℝ≥0∞)
  (hf : ∀ x > 0, Filter.atTop.liminf (f x) ≤ y) :
    ∃ g : ℕ → ℝ≥0, (∀ x, g x > 0) ∧ Filter.atTop.Tendsto g (𝓝 0) ∧
      Filter.atTop.liminf (fun n ↦ f (g n) n) ≤ y
```

lemma exists_liminf_zero_of_forall_liminf_le_with_UB [source]

```
lemma exists_liminf_zero_of_forall_liminf_le_with_UB (y : ℝ≥0) (f : ℝ≥0 → ℕ → ℝ≥0∞)
  {z : ℝ≥0} (hz : 0 < z)
  (hf : ∀ x, x > 0 → Filter.atTop.liminf (f x) ≤ y) :
    ∃ g : ℕ → ℝ≥0, (∀ x, g x > 0) ∧ (∀ x, g x < z) ∧ (Filter.atTop.Tendsto g (𝓝 0)) ∧
      (Filter.atTop.liminf (fun n ↦ f (g n) n) ≤ y)
```

lemma exists_limsup_zero_of_forall_limsup_le [source]

```
lemma exists_limsup_zero_of_forall_limsup_le (y : ℝ≥0) (f : ℝ≥0 → ℕ → ℝ≥0∞)
  (hf : ∀ x, x > 0 → Filter.atTop.limsup (f x) ≤ y) :
    ∃ g : ℕ → ℝ≥0, (∀ x, g x > 0) ∧ (Filter.atTop.Tendsto g (𝓝 0)) ∧
      (Filter.atTop.limsup (fun n ↦ f (g n) n) ≤ y)
```

lemma tendsto_of_block_sequence [source]

```
lemma tendsto_of_block_sequence {α : Type*} [TopologicalSpace α] {x : ℕ → α} {T : ℕ → ℕ}
    (hT : StrictMono T) {L : α} (hx : Filter.atTop.Tendsto x (𝓝 L)) (g : ℕ → α) (hg : ∀ k, ∀ n
     ∈ Set.Ico (T k) (T (k + 1)), g n = x k) :
      Filter.atTop.Tendsto g (𝓝 L)
```

lemma exists_increasing_sequence_with_property [source]

```
lemma exists_increasing_sequence_with_property (M : ℕ → ℕ) (P : ℕ → ℕ → Prop) (hP : ∀ k L, ∃ n
    ≥ L, P k n) :
    ∃ T : ℕ → ℕ, StrictMono T ∧ (∀ k, T k ≥ M k) ∧ (∀ k, ∃ n ∈ Set.Ico (T k) (T (k + 1)), P k n
    )
```

lemma liminf_le_of_block_sequence_witnesses [source]

```
lemma liminf_le_of_block_sequence_witnesses {α : Type*} (y : ℝ≥0) (f : α → ℕ → ℝ≥0∞)
    (T : ℕ → ℕ) (hT : StrictMono T) (x : ℕ → α) (g : ℕ → α)
    (hg : ∀ k, ∀ n ∈ Set.Ico (T k) (T (k + 1)), g n = x k)
    (hwit : ∀ k, ∃ n ∈ Set.Ico (T k) (T (k + 1)), f (x k) n ≤ (y : ℝ≥0∞) + (k + 1 : ℝ≥0)⁻¹) :
    Filter.atTop.liminf (fun n ↦ f (g n) n) ≤ y
```

lemma limsup_le_of_block_sequence_bound [source]

```
lemma limsup_le_of_block_sequence_bound {α : Type*} (y : ℝ≥0) (f : α → ℕ → ℝ≥0∞)
  (T : ℕ → ℕ) (hT : StrictMono T) (x : ℕ → α) (g : ℕ → α)
  (hg : ∀ k, ∀ n ∈ Set.Ico (T k) (T (k + 1)), g n = x k)
  (hbound : ∀ k, ∀ n ∈ Set.Ico (T k) (T (k + 1)), f (x k) n ≤ (y : ℝ≥0∞) + (k + 1 : ℝ≥0)⁻¹) :
  Filter.atTop.limsup (fun n ↦ f (g n) n) ≤ y
```

lemma exists_liminf_zero_of_forall_liminf_limsup_le_with_UB [source]

```
lemma exists_liminf_zero_of_forall_liminf_limsup_le_with_UB (y₁ y₂ : ℝ≥0) (f₁ f₂ : ℝ≥0 → ℕ →
    ℝ≥0∞)
    {z : ℝ≥0} (hz : 0 < z)
    (hf₁ : ∀ x > 0, Filter.atTop.liminf (f₁ x) ≤ y₁)
    (hf₂ : ∀ x > 0, Filter.atTop.limsup (f₂ x) ≤ y₂) :
      ∃ g : ℕ → ℝ≥0, (∀ x, g x > 0) ∧ (∀ x, g x < z) ∧
        Filter.atTop.Tendsto g (𝓝 0) ∧
        Filter.atTop.liminf (fun n ↦ f₁ (g n) n) ≤ y₁ ∧
      Filter.atTop.limsup (fun n ↦ f₂ (g n) n) ≤ y₂
```

theorem extracted_limsup_inequality [source]

```
theorem extracted_limsup_inequality (z : ℝ≥0∞) (hz : z ≠ ⊤) (y x : ℕ → ℝ≥0∞) (h_lem5 : ∀ (n :
    ℕ), x n ≤ y n + z)
    : Filter.atTop.limsup (fun n ↦ x n / n) ≤ Filter.atTop.limsup (fun n ↦ y n / n)
```

theorem tendsto_sub_atTop_iff_nat [source]

Like `Filter.tendsto_add_atTop_iff_nat`, but with nat subtraction.

```
theorem _root_.Filter.tendsto_sub_atTop_iff_nat {α : Type*} {f : ℕ → α} {l : Filter α} (k : ℕ)
    :
    Tendsto (fun (n : ℕ) ↦ f (n - k)) atTop l ↔ Tendsto f atTop l
```

theorem tendsto_sub_const_nhds_zero_iff [source]

Sort of dual to `ENNReal.tendsto_const_sub_nhds_zero_iff`. Takes a substantially different form though, since we don't actually have equality of the limits, or even the fact that the other one converges, which is why we need to use `limsup`.

```
theorem _root_.ENNReal.tendsto_sub_const_nhds_zero_iff {α : Type*} {l : Filter α} {f : α → ℝ≥0
    ∞} {a : ℝ≥0∞}
    : Tendsto (f · - a) l (𝓝 0) ↔ limsup f l ≤ a
```

2.73 QuantumInfo.ForMathlib.LinearEquiv

QuantumInfo/ForMathlib/LinearEquiv.lean on GitHub.

def of_relabel [source]

```
@[simps]
def of_relabel (e : d ≃ d₂) : (d₂ → R) ≃ₗ[R] (d → R)
```

def euclidean_of_relabel [source]

```
@[simps!]
def euclidean_of_relabel (e : d ≃ d₂) : EuclideanSpace 𝕜 d₂ ≃ₗ[𝕜] EuclideanSpace 𝕜 d
```

theorem of_relabel_refl [source]

```
@[simp]
theorem of_relabel_refl : of_relabel R (.refl d) = LinearEquiv.refl R (d → R)
```

theorem euclidean_of_relabel_refl [source]

```
@[simp]
theorem euclidean_of_relabel_refl : euclidean_of_relabel 𝕜 (.refl d) =
    LinearEquiv.refl 𝕜 (EuclideanSpace 𝕜 d)
```

theorem reindex_toLin' [source]

```
theorem reindex_toLin' (e : d₁ ≃ d₃) (f : d₂ ≃ d) (M : Matrix d₁ d₂ R) :
    (M.reindex e f).toLin' = (LinearEquiv.of_relabel R e.symm) ∘ₗ
      M.toLin' ∘ₗ (LinearEquiv.of_relabel R f)
```

theorem reindex_toEuclideanLin [source]

```
theorem reindex_toEuclideanLin (e : d₁ ≃ d₃) (f : d₂ ≃ d) (M : Matrix d₁ d₂ 𝕜) :
    (M.reindex e f).toEuclideanLin = (LinearEquiv.euclidean_of_relabel 𝕜 e.symm) ∘ₗ
      M.toEuclideanLin ∘ₗ (LinearEquiv.euclidean_of_relabel 𝕜 f)
```

theorem reindex_right_toLin' [source]

```
theorem reindex_right_toLin' (e : d ≃ d₂) (M : Matrix d₃ d R) :
    (M.reindex (.refl d₃) e).toLin' = M.toLin' ∘ₗ (LinearEquiv.of_relabel R e)
```

theorem reindex_right_toEuclideanLin [source]

```
theorem reindex_right_toEuclideanLin (e : d ≃ d₂) (M : Matrix d₃ d 𝕜) :
    (M.reindex (.refl d₃) e).toEuclideanLin =
      M.toEuclideanLin ∘ₗ (LinearEquiv.euclidean_of_relabel 𝕜 e)
```

theorem reindex_left_toLin' [source]

```
theorem reindex_left_toLin' (e : d₁ ≃ d₃) (M : Matrix d₁ d₂ R) :
    (M.reindex e (.refl d₂)).toLin' = (LinearEquiv.of_relabel R e.symm) ∘ M.toLin'
```

theorem reindex_left_toEuclideanLin [source]

```
theorem reindex_left_toEuclideanLin (e : d₁ ≃ d₃) (M : Matrix d₁ d₂ 𝕜) :
    (M.reindex e (.refl d₂)).toEuclideanLin =
      (LinearEquiv.euclidean_of_relabel 𝕜 e.symm) ∘ M.toEuclideanLin
```

2.74 QuantumInfo.ForMathlib.Majorization

QuantumInfo/ForMathlib/Majorization.lean on GitHub.

def singularValues [source]

The singular values of a square complex matrix `A`, defined as the square roots of the eigenvalues of `A†A`. These are indexed by `d` without a particular ordering.

Note: We use `A.conjTranspose` as the argument to `isHermitian_mul_conjTranspose_self` so that the underlying Hermitian matrix is `A† * A` (matching the convention in `schattenNorm`).

```
def singularValues (A : Matrix d d ℂ) : d → ℝ
```

lemma singularValues_nonneg [source]

Singular values are nonneg.

```
lemma singularValues_nonneg (A : Matrix d d ℂ) (i : d) :
    0 ≤ singularValues A i
```

def singularValuesSorted [source]

The sorted singular values of a square complex matrix, in decreasing order, indexed by `Fin (Fintype.card d)`.

We define them by sorting the multiset of singular values.

```
noncomputable def singularValuesSorted (A : Matrix d d ℂ) :
    Fin (Fintype.card d) → ℝ
```

lemma singularValuesSorted_nonneg [source]

Sorted singular values are nonneg.

```
lemma singularValuesSorted_nonneg (A : Matrix d d ℂ) (i : Fin (Fintype.card d)) :
    0 ≤ singularValuesSorted A i
```

lemma sum_singularValues_rpow_eq_sum_sorted [source]

The sum ∑ `singularValues A i ^ p` equals the sum over sorted singular values.

```
lemma sum_singularValues_rpow_eq_sum_sorted (A : Matrix d d ℂ) (p : ℝ) :
    ∑ i : d, singularValues A i ^ p =
    ∑ i : Fin (Fintype.card d), singularValuesSorted A i ^ p
```

lemma singularValuesSorted_antitone [source]

Sorted singular values are antitone (decreasing).

```
lemma singularValuesSorted_antitone (A : Matrix d d ℂ) :
    Antitone (singularValuesSorted A)
```

lemma antitone_mul_of_antitone_nonneg [source]

The product of nonneg antitone sequences is antitone.

```
lemma antitone_mul_of_antitone_nonneg {n : ℕ}
    {f g : Fin n → ℝ} (hf : Antitone f) (hg : Antitone g)
    (hf_nn : ∀ i, 0 ≤ f i) (hg_nn : ∀ i, 0 ≤ g i) :
    Antitone (fun i => f i * g i)
```

instance fintypeLinearOrderClassical [source]

A `LinearOrder` on any `Fintype`, obtained classically via well-ordering.

```
scoped instance fintypeLinearOrderClassical (α : Type*) [Fintype α] [DecidableEq α] :
    LinearOrder α
```

def compoundMatrix [source]

The `k`-th compound (exterior-power) matrix of `M`.

```
noncomputable def compoundMatrix (M : Matrix d d ℂ) (k : ℕ) :
    Matrix {S : Finset d // S.card = k} {S : Finset d // S.card = k} ℂ
```

lemma cauchyBinet [source]

***Cauchy–Binet formula** for rectangular matrices: if `A` is `m × n` and `B` is `n × m`, then `det(A * B) = ∑_S det(A[:,S]) * det(B[S,:])` where the sum is over `m`-element subsets `S` of the column/row index.*

```
lemma cauchyBinet {m : ℕ} {n : Type*} [Fintype n] [DecidableEq n] [LinearOrder n]
    {R : Type*} [CommRing R]
    (A : Matrix (Fin m) n R) (B : Matrix n (Fin m) R) :
    (A * B).det = ∑ S : {S : Finset n // S.card = m},
      (A.submatrix id (S.1.orderEmbOfFin S.2)).det *
      (B.submatrix (S.1.orderEmbOfFin S.2) id).det
```

lemma compoundMatrix_mul [source]

The `compoundMatrix` of a product is the product of the `compoundMatrix`s.

```
lemma compoundMatrix_mul (M N : Matrix d d ℂ) (k : ℕ) :
    compoundMatrix (M * N) k = compoundMatrix M k * compoundMatrix N k
```

lemma compoundMatrix_conjTranspose [source]

`compoundMatrix` commutes with `conjTranspose`.

```
lemma compoundMatrix_conjTranspose (M : Matrix d d ℂ) (k : ℕ) :
    compoundMatrix M.conjTranspose k = (compoundMatrix M k).conjTranspose
```

lemma compoundMatrix_diagonal [source]

The compound matrix of a diagonal matrix is diagonal, with entries being products of eigenvalues over k-subsets.

```
lemma compoundMatrix_diagonal (f : d → ℂ) (k : ℕ) :
    compoundMatrix (Matrix.diagonal f) k =
    Matrix.diagonal (fun S : {S : Finset d // S.card = k} =>
      ∏ i : Fin k, f (S.1.orderEmbOfFin S.2 i))
```

lemma singularValues_compoundMatrix_eq [source]

The eigenvalues of the compound matrix of a Hermitian matrix are the products of eigenvalues over k-subsets. More precisely, the singular values of `compoundMatrix M k` are the square roots of products of eigenvalues of M†M over k-subsets.

```
lemma singularValues_compoundMatrix_eq (M : Matrix d d ℂ) (k : ℕ) :
    ∀ (S : {S : Finset d // S.card = k}),
    ∃ (j : {S : Finset d // S.card = k}),
    singularValues (compoundMatrix M k) j =
    ∏ i : Fin k, singularValues M (S.1.orderEmbOfFin S.2 i)
```

lemma prod_le_prod_sorted [source]

The product of nonneg values over a k-subset is at most the product of the k largest values.

```
lemma prod_le_prod_sorted {n : ℕ} {f : Fin n → ℝ}
    (hf : Antitone f) (hf_nn : ∀ i, 0 ≤ f i)
    (k : ℕ) (hk : k ≤ n)
    (g : Fin k → Fin n) (hg : Function.Injective g) :
    ∏ i : Fin k, f (g i) ≤ ∏ i : Fin k, f ⟨i.val, by omega⟩
```

lemma singularValuesSorted_zero_eq_sup [source]

The 0th sorted singular value is the maximum of the singular values.

```
lemma singularValuesSorted_zero_eq_sup {e : Type*} [Fintype e] [DecidableEq e]
    (A : Matrix e e ℂ) (h : 0 < Fintype.card e) :
    singularValuesSorted A ⟨0, h⟩ = Finset.sup' Finset.univ
      (Finset.univ_nonempty_iff.mpr (Fintype.card_pos_iff.mp h))
      (singularValues A)
```

lemma singularValues_mem_sorted [source]

Each singular value appears in the sorted list.

```
lemma singularValues_mem_sorted {e : Type*} [Fintype e] [DecidableEq e]
    (A : Matrix e e ℂ) (i : e) :
    ∃ j : Fin (Fintype.card e), singularValues A i = singularValuesSorted A j
```

lemma singularValuesSorted_mem_values [source]

Each sorted singular value appears among the original singular values.

```
lemma singularValuesSorted_mem_values {e : Type*} [Fintype e] [DecidableEq e]
    (A : Matrix e e ℂ) (j : Fin (Fintype.card e)) :
    ∃ i : e, singularValuesSorted A j = singularValues A i
```

lemma singularValues_compoundMatrix_perm [source]

Stronger version of `singularValues_compoundMatrix_eq` that exposes the permutation.

```
lemma singularValues_compoundMatrix_perm (M : Matrix d d ℂ) (k : ℕ) :
    ∃ σ : {S : Finset d // S.card = k} ≃ {S : Finset d // S.card = k},
    ∀ S, singularValues (compoundMatrix M k) (σ S) =
    ∏ i : Fin k, singularValues M (S.1.orderEmbOfFin S.2 i)
```

lemma singularValues_compoundMatrix_rev [source]

Converse of `singularValues_compoundMatrix_eq`: every singular value of the compound matrix is a product of singular values of M over some k-subset.

```
lemma singularValues_compoundMatrix_rev (M : Matrix d d ℂ) (k : ℕ)
    (j : {S : Finset d // S.card = k}) :
    ∃ (S : {S : Finset d // S.card = k}),
    singularValues (compoundMatrix M k) j =
    ∏ i : Fin k, singularValues M (S.1.orderEmbOfFin S.2 i)
```

lemma exists_sorting_equiv [source]

There exists a bijection `σ : Fin (card d) ≃ d` such that `singularValues M (σ i) = singularValuesSorted M i` for all `i`.

```
lemma exists_sorting_equiv (M : Matrix d d ℂ) :
    ∃ σ : Fin (Fintype.card d) ≃ d,
    ∀ i, singularValues M (σ i) = singularValuesSorted M i
```

lemma prod_singularValues_subset_le_sorted_prod [source]

For any k-subset S of d, the product of singular values over S is $\leq$ the product of the top k sorted singular values.

```
lemma prod_singularValues_subset_le_sorted_prod (M : Matrix d d ℂ) (k : ℕ)
    (hk : k ≤ Fintype.card d) (S : {S : Finset d // S.card = k}) :
    ∏ i : Fin k, singularValues M (S.1.orderEmbOfFin S.2 i) ≤
    ∏ i : Fin k, singularValuesSorted M ⟨i.val, by omega⟩
```

lemma exists_subset_prod_eq_sorted_prod [source]

```
lemma exists_subset_prod_eq_sorted_prod (M : Matrix d d ℂ) (k : ℕ)
    (hk : k ≤ Fintype.card d) :
    ∃ S : {S : Finset d // S.card = k},
    ∏ i : Fin k, singularValues M (S.1.orderEmbOfFin S.2 i) =
    ∏ i : Fin k, singularValuesSorted M ⟨i.val, by omega⟩
```

lemma prod_singularValuesSorted_eq_compoundSV [source]

```
lemma prod_singularValuesSorted_eq_compoundSV (M : Matrix d d ℂ) (k : ℕ)
    (hk : k ≤ Fintype.card d) :
    ∏ i : Fin k, singularValuesSorted M ⟨i.val, by omega⟩ =
    singularValuesSorted (compoundMatrix M k) ⟨0, by
      have : Fintype.card {S : Finset d // S.card = k} = (Fintype.card d).choose k := by
        simp [Fintype.card_subtype]
```

```
      rw [this]; exact Nat.choose_pos hk⟩
```

lemma inner_le_sup_eigenvalue_mul_inner [source]

*The **Rayleigh quotient bound**: For a Hermitian matrix H with eigenvalues λ, we have `v† H v ≤ (max λ) · v† v` for all v.*

```
lemma IsHermitian.inner_le_sup_eigenvalue_mul_inner
    {e : Type*} [Fintype e] [DecidableEq e]
    (H : Matrix e e ℂ) (hH : H.IsHermitian)
    (he : 0 < Fintype.card e)
    (v : e → ℂ) :
    Complex.re (star v ⬝ᵥ H.mulVec v) ≤
    Finset.sup' Finset.univ
      (Finset.univ_nonempty_iff.mpr (Fintype.card_pos_iff.mp he))
      hH.eigenvalues * Complex.re (star v ⬝ᵥ v)
```

lemma eigenvalue_le_singularValuesSorted_sq [source]

```
lemma eigenvalue_le_singularValuesSorted_sq {e : Type*} [Fintype e] [DecidableEq e]
    (A : Matrix e e ℂ) (h : 0 < Fintype.card e) (i : e) :
  (isHermitian_mul_conjTranspose_self A.conjTranspose).eigenvalues i ≤
    (singularValuesSorted A ⟨0, h⟩) ^ 2
```

lemma quadratic_form_le_singularValuesSorted_sq [source]

*The quadratic form of `A† A` is bounded by (max singular value)^2 * $\|v\|^2$.*

```
lemma quadratic_form_le_singularValuesSorted_sq {e : Type*} [Fintype e] [DecidableEq e]
    (A : Matrix e e ℂ) (h : 0 < Fintype.card e) (v : e → ℂ) :
  Complex.re (star v ⬝ᵥ (A.conjTranspose * A).mulVec v) ≤
    (singularValuesSorted A ⟨0, h⟩) ^ 2 * Complex.re (star v ⬝ᵥ v)
```

lemma singularValuesSorted_mul_le [source]

The largest singular value of a matrix product is at most the product of the largest singular values: `σ₁(M * N) ≤ σ₁(M) * σ₁(N)`. This is operator-norm submultiplicativity.

```
lemma singularValuesSorted_mul_le {e : Type*} [Fintype e] [DecidableEq e]
    (M N : Matrix e e ℂ) (h : 0 < Fintype.card e) :
    singularValuesSorted (M * N) ⟨0, h⟩ ≤
    singularValuesSorted M ⟨0, h⟩ * singularValuesSorted N ⟨0, h⟩
```

lemma horn_weak_log_majorization [source]

Horn's inequality (weak log-majorization of singular values): For all `k`, `∏_{i<k} σ↓ᵢ(AB) ≤ ∏_{i<k} σ↓ᵢ(A) · σ↓ᵢ(B)`. This follows from submultiplicativity of the operator norm applied to exterior powers of the matrices.

```
lemma horn_weak_log_majorization (A B : Matrix d d ℂ) (k : ℕ)
    (hk : k ≤ Fintype.card d) :
    ∏ i : Fin k, singularValuesSorted (A * B) ⟨i.val, by omega⟩ ≤
    ∏ i : Fin k, (singularValuesSorted A ⟨i.val, by omega⟩ *
                  singularValuesSorted B ⟨i.val, by omega⟩)
```

lemma rpow_antitone_of_nonneg_antitone [source]

Raising nonneg antitone sequences to a positive power preserves antitonicity.

```
lemma rpow_antitone_of_nonneg_antitone {n : ℕ}
    {f : Fin n → ℝ} (hf : Antitone f) (hf_nn : ∀ i, 0 ≤ f i)
    {r : ℝ} (hr : 0 < r) :
    Antitone (fun i => f i ^ r)
```

lemma rpow_preserves_weak_log_maj [source]

Weak log-majorization is preserved under positive powers.

```
lemma rpow_preserves_weak_log_maj {n : ℕ}
    {x y : Fin n → ℝ}
    (hx_nn : ∀ i, 0 ≤ x i) (hy_nn : ∀ i, 0 ≤ y i)
    (h_log_maj : ∀ (k : ℕ) (_ : k ≤ n),
      ∏ i : Fin k, x ⟨i.val, by omega⟩ ≤
      ∏ i : Fin k, y ⟨i.val, by omega⟩)
    {r : ℝ} (hr : 0 < r) :
    ∀ (k : ℕ) (_ : k ≤ n),
      ∏ i : Fin k, (fun j => x j ^ r) ⟨i.val, by omega⟩ ≤
      ∏ i : Fin k, (fun j => y j ^ r) ⟨i.val, by omega⟩
```

lemma sum_mul_log_nonneg_of_weak_log_maj [source]

```
lemma sum_mul_log_nonneg_of_weak_log_maj {n : ℕ}
    {x y : Fin n → ℝ}
    (hx_pos : ∀ i, 0 < x i) (hy_pos : ∀ i, 0 < y i)
    (hx_anti : Antitone x)
    (h_log_maj : ∀ (k : ℕ) (_ : k ≤ n),
      ∏ i : Fin k, x ⟨i.val, by omega⟩ ≤
      ∏ i : Fin k, y ⟨i.val, by omega⟩) :
    0 ≤ ∑ i, x i * Real.log (y i / x i)
```

lemma sub_ge_mul_log_div [source]

```
lemma sub_ge_mul_log_div {a b : ℝ} (ha : 0 < a) (hb : 0 < b) :
    b - a ≥ a * Real.log (b / a)
```

lemma weak_log_maj_sum_le [source]

```
lemma weak_log_maj_sum_le {n : ℕ}
    {x y : Fin n → ℝ}
    (hx_nn : ∀ i, 0 ≤ x i) (hy_nn : ∀ i, 0 ≤ y i)
    (hx_anti : Antitone x) (hy_anti : Antitone y)
    (h_log_maj : ∀ (k : ℕ) (_ : k ≤ n),
      ∏ i : Fin k, x ⟨i.val, by omega⟩ ≤
      ∏ i : Fin k, y ⟨i.val, by omega⟩) :
    ∑ i, x i ≤ ∑ i, y i
```

lemma weak_log_maj_sum_rpow_le [source]

Weak log-majorization of nonneg antitone sequences implies the sum of powers inequality.

```
lemma weak_log_maj_sum_rpow_le {n : ℕ}
    {x y : Fin n → ℝ}
    (hx_nn : ∀ i, 0 ≤ x i) (hy_nn : ∀ i, 0 ≤ y i)
    (hx_anti : Antitone x) (hy_anti : Antitone y)
    (h_log_maj : ∀ (k : ℕ) (_ : k ≤ n),
      ∏ i : Fin k, x ⟨i.val, by omega⟩ ≤
      ∏ i : Fin k, y ⟨i.val, by omega⟩)
    {r : ℝ} (hr : 0 < r) :
    ∑ i, x i ^ r ≤ ∑ i, y i ^ r
```

lemma sum_rpow_singularValues_mul_le [source]

```
lemma sum_rpow_singularValues_mul_le (A B : Matrix d d ℂ) {r : ℝ} (hr : 0 < r) :
    ∑ i : Fin (Fintype.card d), singularValuesSorted (A * B) i ^ r ≤
    ∑ i : Fin (Fintype.card d),
      (singularValuesSorted A i ^ r * singularValuesSorted B i ^ r)
```

lemma holder_step_for_singularValues [source]

The finite-sum Hölder inequality applied to sequences of r-th powers of sorted singular values.

With conjugate exponents `p' = p/r > 1` and `q' = q/r > 1` (which satisfy `1/p' + 1/q' = 1` when `1/r = 1/p + 1/q`), this gives: `∑ σ↓ᵢ(A)^r · σ↓ᵢ(B)^r ≤ (∑ σ↓ᵢ(A)^p)^{r/p} · (∑ σ↓ᵢ(B)^q)^{r/q}`

Note: The sums on the RHS don't depend on the ordering, so we can replace sorted singular values with unsorted ones.

```
lemma holder_step_for_singularValues (A B : Matrix d d ℂ)
    {r p q : ℝ} (hr : 0 < r) (hp : 0 < p) (hq : 0 < q)
    (hpqr : 1 / r = 1 / p + 1 / q) :
    (∑ i : Fin (Fintype.card d),
      (singularValuesSorted A i ^ r * singularValuesSorted B i ^ r)) ≤
    (∑ i : Fin (Fintype.card d), singularValuesSorted A i ^ p) ^ (r / p) *
    (∑ i : Fin (Fintype.card d), singularValuesSorted B i ^ q) ^ (r / q)
```

2.75 QuantumInfo.ForMathlib.Matrix

QuantumInfo/ForMathlib/Matrix.lean on GitHub.

theorem zero_rank_eq_zero [source]

```
theorem zero_rank_eq_zero {A : Matrix n n 𝕜} [Fintype n] (hA : A.rank = 0) : A = 0
```

theorem smul_selfAdjoint [source]

```
theorem smul_selfAdjoint {c : 𝕜} (hc : _root_.IsSelfAdjoint c) : (c • A).IsHermitian
```

theorem smul_im_zero [source]

```
theorem smul_im_zero {c : 𝕜} (h : RCLike.im c = 0) : (c • A).IsHermitian
```

theorem smul_real [source]

```
theorem smul_real (c : ℝ) : (c • A).IsHermitian
```

def HermitianSubspace [source]

```
def HermitianSubspace (n 𝕜 : Type*) [Fintype n] [RCLike 𝕜] : Subspace ℝ (Matrix n n 𝕜) where
```

theorem re_trace_eq_trace [source]

```
@[simp]
theorem re_trace_eq_trace : RCLike.re (A.trace) = A.trace
```

theorem sum_eigenvalues_eq_trace [source]

The sum of the eigenvalues of a Hermitian matrix is equal to its trace.

```
theorem sum_eigenvalues_eq_trace : ∑ i, hA.eigenvalues i = A.trace
```

theorem eigenvalues_zero_eq_zero [source]

If all eigenvalues are equal to zero, then the matrix is zero.

```
theorem eigenvalues_zero_eq_zero (h : ∀ i, hA.eigenvalues i = 0) : A = 0
```

theorem kroneckerMap_conjTranspose [source]

```
theorem kroneckerMap_conjTranspose : (A ⊗ₖ B)ᴴ = (Aᴴ ⊗ₖ Bᴴ)
```

theorem kroneckerMap_IsHermitian [source]

```
theorem kroneckerMap_IsHermitian : (A ⊗ₖ B).IsHermitian
```

theorem trace_zero [source]

```
theorem trace_zero : A.trace = 0 → A = 0
```

theorem trace_zero_iff [source]

```
@[simp]
theorem trace_zero_iff : A.trace = 0 ↔ A = 0
```

theorem normSq_eq_conj_mul_self [source]

```
theorem _root_.RCLike.normSq_eq_conj_mul_self {z : 𝕜} : RCLike.normSq z = conj z * z
```

theorem sum_eq_ite [source]

```
theorem Finsupp.sum_eq_ite
      {α : Type u_1} {M : Type u_8} {N : Type u_10} [Zero M] [AddCommMonoid N] [Fintype α]
      [DecidableEq M] (f : α →₀ M) (g : α → M → N) :
    f.sum g = ∑ i, if f i ≠ 0 then g i (f i) else 0
```

theorem outer_self_conj [source]

```
theorem outer_self_conj (v : n → 𝕜) : PosSemidef (vecMulVec v (conj v))
```

theorem convex_cone [source]

```
theorem convex_cone {c₁ c₂ : 𝕜} (hc₁ : 0 ≤ c₁) (hc₂ : 0 ≤ c₂) : (c₁ • A + c₂ • B).PosSemidef
```

theorem stdBasisMatrix_iff_eq [source]

A standard basis matrix (with a positive entry) is positive semidefinite iff the entry is on the diagonal.

```
theorem stdBasisMatrix_iff_eq (i j : m) {c : 𝕜} (hc : 0 < c) : (single i j c).PosSemidef ↔ i =
    j
```

theorem PosSemidef_kronecker [source]

```
theorem PosSemidef_kronecker : (A ⊗ₖ B).PosSemidef
```

theorem zero_dotProduct_zero_iff [source]

```
theorem zero_dotProduct_zero_iff : (∀ x : m → 𝕜, 0 = star x ⬝ᵥ A.mulVec x) ↔ A = 0
```

theorem pos_smul [source]

```
theorem pos_smul {c : 𝕜} (hA : (c • A).PosSemidef) (hc : 0 < c) : A.PosSemidef
```

theorem zero_posSemidef_neg_posSemidef_iff [source]

```
theorem zero_posSemidef_neg_posSemidef_iff : A.PosSemidef ∧ (-A).PosSemidef ↔ A = 0
```

theorem toLin_ker_eq_bot [source]

```
theorem toLin_ker_eq_bot (hA : A.PosDef) : LinearMap.ker A.toLin' = ⊥
```

theorem of_toLin_ker_eq_bot [source]

```
theorem of_toLin_ker_eq_bot (hA : LinearMap.ker A.toLin' = ⊥) (hA₂ : A.PosSemidef) : A.PosDef
```

theorem ker_range_antitone [source]

```
theorem ker_range_antitone {d : Type*} [Fintype d] [DecidableEq d] {A B : Matrix d d ℂ}
  (hA : A.IsHermitian) (hB : B.IsHermitian) :
    LinearMap.ker A.toEuclideanLin ≤ LinearMap.ker B.toEuclideanLin ↔
    LinearMap.range B.toEuclideanLin ≤ LinearMap.range A.toEuclideanLin
```

theorem le_of_nonneg_imp [source]

```
theorem le_of_nonneg_imp {R : Type*} [AddCommGroup R] [PartialOrder R] [IsOrderedAddMonoid R]
    (f : Matrix n n 𝕜 →+ R) (h : ∀ A, A.PosSemidef → 0 ≤ f A) :
    (A ≤ B → f A ≤ f B)
```

theorem le_of_nonneg_imp' [source]

```
theorem le_of_nonneg_imp' {R : Type*} [AddCommGroup R] [PartialOrder R] [IsOrderedAddMonoid R]
    {x y : R} (f : R →+ Matrix n n 𝕜) (h : ∀ x, 0 ≤ x → (f x).PosSemidef) :
    (x ≤ y → f x ≤ f y)
```

theorem mul_mul_conjTranspose_mono [source]

```
theorem mul_mul_conjTranspose_mono (C : Matrix m n 𝕜) :
  A ≤ B → C * A * C.conjTranspose ≤ C * B * C.conjTranspose
```

theorem conjTranspose_mul_mul_mono [source]

```
theorem conjTranspose_mul_mul_mono (C : Matrix n m 𝕜) :
  A ≤ B → C.conjTranspose * A * C ≤ C.conjTranspose * B * C
```

theorem nonneg_iff_eigenvalue_nonneg [source]

```
theorem nonneg_iff_eigenvalue_nonneg [DecidableEq n] : 0 ≤ A ↔ ∀ x, 0 ≤ hA.eigenvalues x
```

theorem diag_monotone [source]

```
theorem diag_monotone : Monotone (diag : Matrix n n 𝕜 → (n → 𝕜))
```

theorem diag_mono [source]

```
theorem diag_mono : A ≤ B → ∀ i, A.diag i ≤ B.diag i
```

theorem trace_monotone [source]

```
theorem trace_monotone : Monotone (@trace n 𝕜 _ _)
```

theorem trace_mono [source]

```
theorem trace_mono : A ≤ B → A.trace ≤ B.trace
```

theorem diagonal_monotone [source]

```
theorem diagonal_monotone : Monotone (diagonal : (n → 𝕜) → _)
```

theorem diagonal_mono [source]

```
theorem diagonal_mono {d₁ d₂ : n → 𝕜} : d₁ ≤ d₂ → diagonal d₁ ≤ diagonal d₂
```

theorem diagonal_le_iff [source]

```
theorem diagonal_le_iff {d₁ d₂ : n → 𝕜} : d₁ ≤ d₂ ↔ diagonal d₁ ≤ diagonal d₂
```

theorem le_smul_one_of_eigenvalues_iff [source]

```
theorem le_smul_one_of_eigenvalues_iff (hA : A.IsHermitian) (c : ℝ) :
  (∀ i, hA.eigenvalues i ≤ c) ↔ A ≤ c • (1 : Matrix n n 𝕜)
```

theorem smul_one_le_of_eigenvalues_iff [source]

```
theorem smul_one_le_of_eigenvalues_iff (hA : A.IsHermitian) (c : ℝ) :
  (∀ i, c ≤ hA.eigenvalues i) ↔ c • (1 : Matrix n n 𝕜) ≤ A
```

def Matrix.traceLeft [source]

```
def traceLeft (m : Matrix (d × d₁) (d × d₂) R) : Matrix d₁ d₂ R
```

def Matrix.traceRight [source]

```
def traceRight (m : Matrix (d₁ × d) (d₂ × d) R) : Matrix d₁ d₂ R
```

theorem traceLeft_trace [source]

```
@[simp]
theorem traceLeft_trace (A : Matrix (d₁ × d₂) (d₁ × d₂) R) : A.traceLeft.trace = A.trace
```

theorem traceRight_trace [source]

```
@[simp]
theorem traceRight_trace (A : Matrix (d₁ × d₂) (d₁ × d₂) R) : A.traceRight.trace = A.trace
```

theorem IsHermitian.traceLeft [source]

```
theorem IsHermitian.traceLeft {A : Matrix (d × d₁) (d × d₁) R} (hA : A.IsHermitian) : A.
    traceLeft.IsHermitian
```

theorem IsHermitian.traceRight [source]

```
theorem IsHermitian.traceRight {A : Matrix (d₁ × d) (d₁ × d) R} (hA : A.IsHermitian) : A.
    traceRight.IsHermitian
```

theorem trace_mul_kron_one_right [source]

Tr(M (A ⊗ I)) = Tr(Tr_B(M) A)

```
theorem trace_mul_kron_one_right {R : Type*} [Ring R]
    (M : Matrix (dA × dB) (dA × dB) R) (A : Matrix dA dA R) :
    (M * (A ⊗ₖ (1 : Matrix dB dB R))).trace = (M.traceRight * A).trace
```

theorem trace_mul_one_kron_right [source]

Tr(M (I ⊗ B)) = Tr(Tr_A(M) B)

```
theorem trace_mul_one_kron_right {R : Type*} [Ring R]
    (M : Matrix (dA × dB) (dA × dB) R) (B : Matrix dB dB R) :
    (M * ((1 : Matrix dA dA R) ⊗ₖ B)).trace = (M.traceLeft * B).trace
```

theorem PosSemidef.traceLeft [source]

```
theorem PosSemidef.traceLeft [DecidableEq d₁] (hA : A.PosSemidef) : A.traceLeft.PosSemidef
```

theorem PosSemidef.traceRight [source]

```
theorem PosSemidef.traceRight [DecidableEq d₂] (hA : A.PosSemidef) : A.traceRight.PosSemidef
```

theorem kron [source]

```
theorem PosDef.kron {d₁ d₂ 𝕜 : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [DecidableEq d
    ₂] [RCLike 𝕜]
    {A : Matrix d₁ d₁ 𝕜} {B : Matrix d₂ d₂ 𝕜} (hA : A.PosDef) (hB : B.PosDef) : (A ⊗ₖ B).PosDef
```

theorem submatrix [source]

```
theorem PosDef.submatrix {d₁ d₂ 𝕜 : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [
    DecidableEq d₂] [RCLike 𝕜]
    {M : Matrix d₁ d₁ 𝕜} (hM : M.PosDef) {e : d₂ → d₁} (he : Function.Injective e) : (M.
    submatrix e e).PosDef
```

theorem reindex [source]

```
theorem PosDef.reindex {d₁ d₂ 𝕜 : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [
    DecidableEq d₂] [RCLike 𝕜]
    {M : Matrix d₁ d₁ 𝕜} (hM : M.PosDef) (e : d₁ ≃ d₂) : (M.reindex e e).PosDef
```

theorem reindex_iff [source]

```
@[simp]
theorem PosDef.reindex_iff {d₁ d₂ 𝕜 : Type*} [Fintype d₁] [DecidableEq d₁] [Fintype d₂] [
    DecidableEq d₂] [RCLike 𝕜]
    {M : Matrix d₁ d₁ 𝕜} (e : d₁ ≃ d₂) : (M.reindex e e).PosDef ↔ M.PosDef
```

theorem rsmul [source]

```
theorem PosSemidef.rsmul {n : Type*} [Fintype n] {M : Matrix n n ℂ} (hM : M.PosSemidef) {c : ℝ}
     (hc : 0 ≤ c) :
    (c • M).PosSemidef
```

theorem Convex [source]

```
theorem PosDef.Convex {n 𝕜 : Type*} [Fintype n] [RCLike 𝕜] : Convex ℝ (Matrix.PosDef (n := n) (
    R := 𝕜))
```

theorem PosDef_iff_eigenvalues' [source]

```
theorem PosDef_iff_eigenvalues' (M : Matrix d d 𝕜) :
    M.PosDef ↔ ∃ (h : M.IsHermitian), ∀ i, 0 < h.eigenvalues i
```

theorem cfc_eigenvalues [source]

```
theorem IsHermitian.cfc_eigenvalues {M : Matrix d d 𝕜} (hM : M.IsHermitian) (f : ℝ → ℝ) :
    ∃ (e : d ≃ d), Matrix.IsHermitian.eigenvalues (cfc_predicate f M) = f ∘ hM.eigenvalues ∘ e
```

lemma eigenvalues_eq_of_unitary_similarity_diagonal [source]

If a Hermitian matrix A is unitarily similar to a diagonal matrix with real entries f, then the eigenvalues of A are a permutation of f.

```
lemma IsHermitian.eigenvalues_eq_of_unitary_similarity_diagonal {d 𝕜 : Type*}
    [Fintype d] [DecidableEq d] [RCLike 𝕜]
    {A : Matrix d d 𝕜} (hA : A.IsHermitian)
    {U : Matrix d d 𝕜} (hU : U ∈ Matrix.unitaryGroup d 𝕜)
    {f : d → ℝ}
    (h : A = U * Matrix.diagonal (fun i => (RCLike.ofReal (f i) : 𝕜)) * Matrix.conjTranspose U)
     :
    ∃ σ : d ≃ d, hA.eigenvalues ∘ σ = f
```

theorem toEuclideanLin_one [source]

```
@[simp]
theorem toEuclideanLin_one : Matrix.toEuclideanLin (1 : Matrix n n α) = .id
```

theorem cfc_diagonal [source]

```
@[simp]
theorem cfc_diagonal (g : d → ℝ) (f : ℝ → ℝ) :
    cfc f (Matrix.diagonal (fun x ↦ (g x : 𝕜))) = diagonal (RCLike.ofReal ∘ f ∘ g)
```

theorem pos_of_mem_spectrum [source]

```
theorem PosSemidef.pos_of_mem_spectrum {A : Matrix d d 𝕜} (hA : A.PosSemidef) (r : ℝ) :
    r ∈ spectrum ℝ A → 0 ≤ r
```

theorem pow_add [source]

```
theorem PosSemidef.pow_add {A : Matrix d d 𝕜} (hA : A.PosSemidef) {x y : ℝ} (hxy : x + y ≠ 0) :
    cfc (· ^ (x + y) : ℝ → ℝ) A = cfc (fun r ↦ r ^ x * r ^ y : ℝ → ℝ) A
```

theorem pow_mul [source]

```
theorem PosSemidef.pow_mul {A : Matrix d d 𝕜} {x y : ℝ} (hA : A.PosSemidef) :
    cfc (· ^ (x * y) : ℝ → ℝ) A = cfc (fun r ↦ (r ^ x) ^ y : ℝ → ℝ) A
```

theorem trace_submatrix [source]

```
@[simp]
theorem trace_submatrix
  (A : Matrix d₁ d₁ α) (e : d₂ ≃ d₁) :
    (A.submatrix e e).trace = A.trace
```

lemma spectrum_prod_complex [source]

```
private lemma spectrum_prod_complex {d d₂ : Type*}
  [Fintype d] [DecidableEq d] [Fintype d₂] [DecidableEq d₂]
  {A : Matrix d d 𝕜} {B : Matrix d₂ d₂ 𝕜}
  (hA : A.IsHermitian) (hB : B.IsHermitian) :
    ∀ x : 𝕜, x ∈ spectrum 𝕜 (A ⊗ₖ B) → ∃ a ∈ spectrum 𝕜 A, ∃ b ∈ spectrum 𝕜 B, x = a * b
```

lemma spectrum_prod_le [source]

```
private lemma spectrum_prod_le {d d₂ : Type*}
  [Fintype d] [DecidableEq d] [Fintype d₂] [DecidableEq d₂]
  {A : Matrix d d 𝕜} {B : Matrix d₂ d₂ 𝕜}
  (hA : A.IsHermitian) (hB : B.IsHermitian) :
    spectrum ℝ (A ⊗ₖ B) ⊆ spectrum ℝ A * spectrum ℝ B
```

theorem spectrum_prod [source]

```
theorem spectrum_prod {d d₂ : Type*}
  [Fintype d] [DecidableEq d] [Fintype d₂] [DecidableEq d₂]
  {A : Matrix d d 𝕜} {B : Matrix d₂ d₂ 𝕜}
  (hA : A.IsHermitian) (hB : B.IsHermitian) :
    spectrum ℝ (A ⊗ₖ B) = spectrum ℝ A * spectrum ℝ B
```

theorem zero_lt [source]

```
theorem PosDef.zero_lt {n : Type*} [Nonempty n] [Fintype n] {A : Matrix n n ℂ} (hA : A.PosDef)
    : 0 < A
```

lemma spectrum_eq_image_eigenvalues [source]

```
lemma IsHermitian.spectrum_eq_image_eigenvalues [Fintype n] {A : Matrix n n ℂ} (hA : A.
    IsHermitian) :
    spectrum ℝ A = Finset.univ.image hA.eigenvalues
```

lemma card_spectrum_eq_image [source]

```
lemma IsHermitian.card_spectrum_eq_image [Fintype n] {A : Matrix n n ℂ} (hA : A.IsHermitian)
  [Fintype (spectrum ℝ A)] :
    Fintype.card (spectrum ℝ A) = (Finset.univ.image hA.eigenvalues).card
```

lemma sub_iInf_eignevalues [source]

```
lemma sub_iInf_eignevalues (hA : A.IsHermitian) :
    (A - iInf hA.eigenvalues • 1).PosSemidef
```

lemma iInf_eigenvalues_le_dotProduct_mulVec [source]

```
lemma iInf_eigenvalues_le_dotProduct_mulVec (hA : A.IsHermitian) (v : d → ℂ) :
    iInf hA.eigenvalues * (star v ⬝ᵥ v) ≤ star v ⬝ᵥ A *ᵥ v
```

lemma iInf_eigenvalues_le_of_posSemidef [source]

```
lemma iInf_eigenvalues_le_of_posSemidef
  (hAB : (B - A).PosSemidef) (hA : A.IsHermitian) (hB : B.IsHermitian) :
    iInf hA.eigenvalues ≤ iInf hB.eigenvalues
```

lemma iInf_eigenvalues_le [source]

```
lemma iInf_eigenvalues_le (hAB : A ≤ B) (hA : A.IsHermitian) (hB : B.IsHermitian) :
    iInf hA.eigenvalues ≤ iInf hB.eigenvalues
```

lemma iInf_eigenvalues_smul_one_le [source]

```
lemma iInf_eigenvalues_smul_one_le (hA : A.IsHermitian) : iInf hA.eigenvalues • 1 ≤ A
```

lemma _shortcut_posSMulMono [source]

```
lemma _shortcut_posSMulMono : PosSMulMono ℝ (Matrix d d ℂ)
```

lemma _shortcut_posSmulReflectLE [source]

```
lemma _shortcut_posSmulReflectLE : PosSMulReflectLE ℝ (Matrix d d ℂ)
```

theorem spectrum_subset_Ici_of_sub [source]

```
theorem IsHermitian.spectrum_subset_Ici_of_sub {d 𝕜 : Type*} [Fintype d] [DecidableEq d] [
    RCLike 𝕜]
  {A x : Matrix d d 𝕜} (hA : A.IsHermitian) (hl : (x - A).PosSemidef) :
    spectrum ℝ x ⊆ Set.Ici (⨅ i, hA.eigenvalues i)
```

theorem spectrum_subset_Iic_of_sub [source]

```
theorem IsHermitian.spectrum_subset_Iic_of_sub {d 𝕜 : Type*} [Fintype d] [DecidableEq d] [
    RCLike 𝕜]
  {A x : Matrix d d 𝕜} (hA : A.IsHermitian) (hl : (A - x).PosSemidef) :
    spectrum ℝ x ⊆ Set.Iic (⨆ i, hA.eigenvalues i)
```

theorem spectrum_subset_of_mem_Icc [source]

```
theorem IsHermitian.spectrum_subset_of_mem_Icc {d 𝕜 : Type*} [Fintype d] [DecidableEq d] [
    RCLike 𝕜]
  {A B x : Matrix d d 𝕜} (hA : A.IsHermitian) (hB : B.IsHermitian)
  (hl : (x - A).PosSemidef) (hr : (B - x).PosSemidef) :
    spectrum ℝ x ⊆ Set.Icc (⨅ i, hA.eigenvalues i) (⨆ i, hB.eigenvalues i)
```

theorem traceRight_eq_traceLeft_reindex [source]

The right partial trace of a matrix is equal to the left partial trace of the matrix reindexed by swapping the tensor factors.

```
theorem traceRight_eq_traceLeft_reindex {n m R : Type*} [Fintype m] [AddCommMonoid R]
  (M : Matrix (n × m) (n × m) R) :
    M.traceRight = (M.reindex (.prodComm ..) (.prodComm ..)).traceLeft
```

theorem trace_pos [source]

```
theorem PosSemidef.trace_pos {n 𝕜 : Type*} [Fintype n] [RCLike 𝕜]
    {A : Matrix n n 𝕜} (hA : A.PosSemidef) (h : A ≠ 0) : 0 < A.trace
```

theorem traceLeft_add [source]

```
@[simp]
theorem traceLeft_add : (A + B).traceLeft = A.traceLeft + B.traceLeft
```

theorem traceLeft_neg [source]

```
@[simp]
theorem traceLeft_neg : (-A).traceLeft = -A.traceLeft
```

theorem traceLeft_sub [source]

```
@[simp]
theorem traceLeft_sub : (A - B).traceLeft = A.traceLeft - B.traceLeft
```

theorem traceRight_add [source]

```
@[simp]
theorem traceRight_add : (A + B).traceRight = A.traceRight + B.traceRight
```

theorem traceRight_neg [source]

```
@[simp]
theorem traceRight_neg : (-A).traceRight = -A.traceRight
```

theorem traceRight_sub [source]

```
@[simp]
theorem traceRight_sub : (A - B).traceRight = A.traceRight - B.traceRight
```

theorem traceLeft_smul [source]

```
@[simp]
theorem traceLeft_smul {A : Matrix (m × n) (m × n) α} (r : R) :
    (r • A).traceLeft = r • A.traceLeft
```

theorem traceRight_smul [source]

```
@[simp]
theorem traceRight_smul {A : Matrix (n × m) (n × m) α} (r : R) :
    (r • A).traceRight = r • A.traceRight
```

theorem unitaryGroup_row_norm [source]

```
theorem unitaryGroup_row_norm [Fintype n] (U : Matrix.unitaryGroup n ℂ) (i : n) :
    ∑ j, ‖U j i‖^2 = 1
```

def Matrix.piProd [source]

```
def piProd [CommMonoid R] (A : ∀ i, Matrix (d i) (d i) R) : Matrix (∀ i, d i) (∀ i, d i) R
```

theorem IsHermitian.piProd [source]

```
theorem IsHermitian.piProd [CommSemiring R] [StarRing R] (hA : ∀ i, (A i).IsHermitian) :
    (piProd A).IsHermitian
```

theorem trace_piProd [source]

```
theorem trace_piProd [CommSemiring R] :
    (piProd A).trace = ∏ i, (A i).trace
```

theorem PosSemidef.piProd [source]

```
theorem PosSemidef.piProd [RCLike R] (hA : ∀ i, (A i).PosSemidef) :
    (piProd A).PosSemidef
```

theorem submatrix_eq_mul_mul [source]

```
theorem submatrix_eq_mul_mul {d d₂ d₃ R : Type*} [DecidableEq d] [Fintype d] [Semiring R]
  (A : Matrix d d R) (e : d₂ → d) (f : d₃ → d) :
    A.submatrix e f = (submatrix (α := R) 1 e id : Matrix d₂ d R) * A * (submatrix (α := R) 1
    id f)
```

lemma kronecker_conj_eq [source]

The conjugate of a Kronecker product by a Kronecker product is the Kronecker product of the conjugates (for matrices).

```
lemma kronecker_conj_eq {m n p q α : Type*} [CommSemiring α] [StarRing α] [Fintype m] [Fintype
    n]
    (A : Matrix m m α) (B : Matrix n n α) (C : Matrix p m α) (D : Matrix q n α) :
    (C ⊗ₖ D) * (A ⊗ₖ B) * (C ⊗ₖ D)ᴴ = (C * A * Cᴴ) ⊗ₖ (D * B * Dᴴ)
```

2.76 QuantumInfo.ForMathlib.MatrixNorm.TraceNorm

QuantumInfo/ForMathlib/MatrixNorm/TraceNorm.lean on GitHub.

def traceNorm [source]

The trace norm of a matrix: Tr[√(A† A)].

```
def traceNorm (A : Matrix m n R) : ℝ
```

theorem traceNorm_zero [source]

```
@[simp]
theorem traceNorm_zero : traceNorm (0 : Matrix m n R) = 0
```

theorem traceNorm_eq_neg_self [source]

The trace norm of the negative is equal to the trace norm.

```
@[simp]
theorem traceNorm_eq_neg_self (A : Matrix m n R) : traceNorm (-A) = traceNorm A
```

lemma cfc_sqrt_isometry_conj [source]

```
lemma cfc_sqrt_isometry_conj {A : Matrix n n R} (hA : 0 ≤ A)
  {u : Matrix m n R} (hu₁ : u.Isometry) :
    CFC.sqrt (u * A * u^H) = u * CFC.sqrt A * u^H
```

theorem traceNorm_isometry_left [source]

```
theorem traceNorm_isometry_left [Fintype k] {A : Matrix n m R} {u : Matrix k n R}
  (hu₁ : u.Isometry) : traceNorm (u * A) = traceNorm A
```

theorem traceNorm_isometry_right [source]

```
theorem traceNorm_isometry_right [Fintype k] {A : Matrix n m R} {u : Matrix k m R}
  (hu₁ : u.Isometry) : traceNorm (A * u^H) = traceNorm A
```

theorem traceNorm_isometry_conj [source]

The trace norm is invariant under isometries u and v, Property 9.1.4 in Wilde

```
theorem traceNorm_isometry_conj {A : Matrix n n R} {u : Matrix m n R}
  (hu : u.Isometry) {v : Matrix m n R} (hv : v.Isometry) :
    traceNorm (u * A * v^H) = traceNorm A
```

theorem traceNorm_unitary_conj [source]

```
@[simp]
theorem traceNorm_unitary_conj {A : Matrix n n R} {U : Matrix.unitaryGroup n R} :
  traceNorm (U.val * A * U.val^H) = traceNorm A
```

theorem traceNorm_Hermitian_eq_sum_abs_eigenvalues [source]

```
theorem traceNorm_Hermitian_eq_sum_abs_eigenvalues {A : Matrix n n R} (hA : A.IsHermitian) :
    A.traceNorm = ∑ i, abs (hA.eigenvalues i)
```

theorem traceNorm_nonneg [source]

The trace norm is nonnegative. Property 9.1.1 in Wilde

```
theorem traceNorm_nonneg (A : Matrix m n R) : 0 ≤ A.traceNorm
```

theorem traceNorm_zero_iff [source]

The trace norm is zero iff. the matrix is zero.

```
theorem traceNorm_zero_iff (A : Matrix m n R) : A.traceNorm = 0 ↔ A = 0
```

theorem traceNorm_smul [source]

Trace norm is linear under scalar multiplication. Property 9.1.2 in Wilde

```
theorem traceNorm_smul (A : Matrix m n R) (c : R) : (c • A).traceNorm = ‖c‖ * A.traceNorm
```

theorem traceNorm_eq_max_tr_U [source]

*For square matrices, the trace norm is max Tr[U * A] over unitaries U.*

```
theorem traceNorm_eq_max_tr_U (A : Matrix n n R) : IsGreatest {x | ∃ (U : unitaryGroup n R), (U
    .1 * A).trace = x} A.traceNorm
```

theorem traceNorm_triangleIneq [source]

the trace norm satisfies the triangle inequality (for square matrices). TODO: Prove in general.

```
theorem traceNorm_triangleIneq (A B : Matrix n n R) : (A + B).traceNorm ≤ A.traceNorm + B.
    traceNorm
```

theorem traceNorm_triangleIneq' [source]

```
theorem traceNorm_triangleIneq' (A B : Matrix n n R) : (A - B).traceNorm ≤ A.traceNorm + B.
    traceNorm
```

theorem traceNorm_PSD_eq_trace [source]

```
theorem PosSemidef.traceNorm_PSD_eq_trace {A : Matrix m m R} (hA : A.PosSemidef) : A.traceNorm
    = A.trace
```

theorem traceNorm_convex [source]

The trace norm is convex. Property 9.1.5 in Wilde

```
theorem traceNorm_convex (M N : Matrix n n R) (l : ℝ) (hl : 0 ≤ l ∧ l ≤ 1) :
  ((l:R) • M + ((1 - l) : R) • N).traceNorm ≤ l * M.traceNorm + (1-l) * N.traceNorm
```

2.77 QuantumInfo.ForMathlib.Minimax

QuantumInfo/ForMathlib/Minimax.lean on GitHub.

theorem exists_isMinOn_lowerSemicontinuousOn [source]

```
theorem _root_.IsCompact.exists_isMinOn_lowerSemicontinuousOn {α β : Type*}
  [LinearOrder α] [TopologicalSpace α] [TopologicalSpace β] [ClosedIicTopology α]
  {s : Set β} (hs : IsCompact s) (ne_s : s.Nonempty) {f : β → α} (hf : LowerSemicontinuousOn f
    s) :
    ∃ x ∈ s, IsMinOn f s x
```

theorem quasilinearOn [source]

```
@[fun_prop]
theorem LinearMap.quasilinearOn {E β 𝕜 : Type*} [Semiring 𝕜] [PartialOrder 𝕜]
  [AddCommMonoid E] [Module 𝕜 E]
  [PartialOrder β] [AddCommMonoid β] [IsOrderedAddMonoid β] [Module 𝕜 β] [PosSMulMono 𝕜 β]
  (f : E →ₗ[𝕜] β) {s : Set E} (hs : Convex 𝕜 s) :
    QuasilinearOn 𝕜 s f
```

theorem quasiconvexOn [source]

```
@[fun_prop]
theorem LinearMap.quasiconvexOn {E β 𝕜 : Type*} [Semiring 𝕜] [PartialOrder 𝕜]
  [AddCommMonoid E] [Module 𝕜 E]
  [PartialOrder β] [AddCommMonoid β] [IsOrderedAddMonoid β] [Module 𝕜 β] [PosSMulMono 𝕜 β]
  (f : E →ₗ[𝕜] β) {s : Set E} (hs : Convex 𝕜 s) :
    QuasiconvexOn 𝕜 s f
```

theorem quasiconcaveOn [source]

```
@[fun_prop]
theorem LinearMap.quasiconcaveOn {E β 𝕜 : Type*} [Semiring 𝕜] [PartialOrder 𝕜]
  [AddCommMonoid E] [Module 𝕜 E]
  [PartialOrder β] [AddCommMonoid β] [IsOrderedAddMonoid β] [Module 𝕜 β] [PosSMulMono 𝕜 β]
  (f : E →ₗ[𝕜] β) {s : Set E} (hs : Convex 𝕜 s) :
    QuasiconcaveOn 𝕜 s f
```

theorem [source]

```
theorem continuous_stupid.{u_2, u_1} {M : Type u_1} [inst : NormedAddCommGroup M] [inst_1 :
    Module ℝ M]
  [inst_3 : ContinuousSMul ℝ M] {N : Type u_2} [inst_4 : NormedAddCommGroup N]
  [inst_5 : Module ℝ N]
  [FiniteDimensional ℝ M]
  (f : N →L[ℝ] M →L[ℝ] ℝ) :
    Continuous fun (x : N × M) ↦ (f x.1) x.2
```

theorem minimax [source]

The minimax theorem, at the level of generality we need. For convex, compact, nonempty sets `S` and `T` in a real topological vector space `M`, and a bilinear function `f` on `M`, we can exchange the order of minimizing and maximizing.

```
theorem minimax
  {M : Type*} [NormedAddCommGroup M] [Module ℝ M] [ContinuousAdd M] [ContinuousSMul ℝ M] [
    FiniteDimensional ℝ M]
  {N : Type*} [NormedAddCommGroup N] [Module ℝ N] [ContinuousAdd N] [ContinuousSMul ℝ N]
```

```
  (f : N →L[ℝ] M →L[ℝ] ℝ)
  (S : Set M) (T : Set N) (hS₁ : IsCompact S) (hT₁ : IsCompact T)
  (hS₂ : Convex ℝ S) (hT₂ : Convex ℝ T) (hS₃ : S.Nonempty) (hT₃ : T.Nonempty)
    : ⨆ x : T, ⨅ y : S, f x y = ⨅ y : S, ⨆ x : T, f x y
```

theorem BilinForm.minimax [source]

***Von-Neumann's Minimax Theorem**, specialized to bilinear forms.*

```
theorem LinearMap.BilinForm.minimax
  {M : Type*} [NormedAddCommGroup M] [Module ℝ M] [ContinuousAdd M] [ContinuousSMul ℝ M] [
    FiniteDimensional ℝ M]
  (f : LinearMap.BilinForm ℝ M)
  (S : Set M) (T : Set M) (hS₁ : IsCompact S) (hT₁ : IsCompact T)
  (hS₂ : Convex ℝ S) (hT₂ : Convex ℝ T) (hS₃ : S.Nonempty) (hT₃ : T.Nonempty)
    : ⨆ x : T, ⨅ y : S, f x y = ⨅ y : S, ⨆ x : T, f x y
```

theorem minimax' [source]

Convenience form of `LinearMap.BilinForm.minimax` with the order inf/sup arguments supplied to f flipped.

```
theorem LinearMap.BilinForm.minimax'
  {M : Type*} [NormedAddCommGroup M] [Module ℝ M] [ContinuousAdd M] [ContinuousSMul ℝ M] [
    FiniteDimensional ℝ M]
  (f : LinearMap.BilinForm ℝ M)
  (S : Set M) (T : Set M) (hS₁ : IsCompact S) (hT₁ : IsCompact T)
  (hS₂ : Convex ℝ S) (hT₂ : Convex ℝ T) (hS₃ : S.Nonempty) (hT₃ : T.Nonempty)
    : ⨅ x : S, ⨆ y : T, f x y = ⨆ y : T, ⨅ x : S, f x y
```

2.78 QuantumInfo.ForMathlib.Misc

QuantumInfo/ForMathlib/Misc.lean on GitHub.

theorem ite_eq_top [source]

```
@[aesop (rule_sets := [finiteness]) unsafe apply]
theorem ite_eq_top {α : Type*} [Top α] (h : Prop) [Decidable h] {x y : α} (hx : x ≠ ⊤) (hy : y
    ≠ ⊤) :
    (if h then x else y) ≠ ⊤
```

theorem subtype_val_iSup [source]

```
theorem subtype_val_iSup (h : ∀ i, f i ∈ Set.Icc a b) :
    (⨆ i, ((f i, h i) : ↑(Set.Icc a b))).val = ⨆ i, f i
```

theorem subtype_val_iSup' [source]

```
theorem subtype_val_iSup' (h : ∀ i, f i ∈ Set.Icc a b) :
    ⨆ i, ((f i, h i) : ↑(Set.Icc a b)) =
      ⟨⨆ i, f i, ⟨(h i.some).1.trans (le_ciSup ⟨b, by intro; grind⟩ _), ciSup_le (h ·|>.2)⟩⟩
```

theorem subtype_val_iInf [source]

```
theorem subtype_val_iInf (h : ∀ i, f i ∈ Set.Icc a b) :
    (⨅ i, ((f i, h i) : ↑(Set.Icc a b))).val = ⨅ i, f i
```

theorem subtype_val_iInf' [source]

```
theorem subtype_val_iInf' (h : ∀ i, f i ∈ Set.Icc a b) :
    ⨅ i, ((f i, h i) : ↑(Set.Icc a b)) =
      ⟨⨅ i, f i, ⟨le_ciInf (h ·|>.1), (ciInf_le ⟨a, by intro; grind⟩ _).trans (h i.some).2⟩⟩
```

lemma bdd_le_mul_tendsto_zero [source]

Analogous to `bdd_le_mul_tendsto_zero`, for `ENNReal` (which otherwise lacks a continuous multiplication function). The product of a sequence that tends to zero with any bounded sequence also tends to zero.

```
protected lemma ENNReal.bdd_le_mul_tendsto_zero
  {α : Type*} {l : Filter α} {f g : α → ℝ≥0∞} {b : ℝ≥0∞} (hb : b ≠ ⊤)
  (hf : l.Tendsto f (𝓝 0)) (hg : ∀ᶠ (x : α) in l, g x ≤ b) :
    l.Tendsto (fun x ↦ f x * g x) (𝓝 0)
```

theorem csInf_mul_nonneg [source]

```
theorem csInf_mul_nonneg {s t : Set ℝ}
  (hs₀ : s.Nonempty) (hs₁ : ∀ x ∈ s, 0 ≤ x) (ht₀ : t.Nonempty) (ht₁ : ∀ x ∈ t, 0 ≤ x) :
    sInf (s * t) = sInf s * sInf t
```

lemma map_univ_eq_iff [source]

If two functions from finite types have the same multiset of values, there exists a bijection between the domains that commutes with the functions.

```
lemma Multiset.map_univ_eq_iff {α β : Type*} [Fintype α] (f g : α → β) :
    Multiset.map f Finset.univ.val = Multiset.map g Finset.univ.val ↔ ∃ (e : α ≃ α), f = g ∘ e
```

lemma exists_equiv_of_multiset_map_eq [source]

If two functions from finite types have the same multiset of values, there exists a bijection between the domains that commutes with the functions.

```
lemma exists_equiv_of_multiset_map_eq {α β γ : Type*} [Fintype α] [Fintype β] [DecidableEq γ]
    (f : α → γ) (g : β → γ) (h : Multiset.map f Finset.univ.val = Multiset.map g Finset.univ.
    val) :
    ∃ e : α ≃ β, f = g ∘ e
```

2.79 QuantumInfo.ForMathlib.SionMinimax

QuantumInfo/ForMathlib/SionMinimax.lean on GitHub.

theorem image2_flip [source]

```
@[simp]
theorem Set.image2_flip {α β γ : Type*} {f : α → β → γ} (s : Set α) (t : Set β) :
```

```
    image2 (flip f) t s = image2 f s t
```

theorem ciSup_ciInf_le_ciInf_ciSup [source]

*The **max-min theorem**. A version of `iSup_iInf_le_iInf_iSup` for conditionally complete lattices.*

```
theorem ciSup_ciInf_le_ciInf_ciSup {ι': Type*} [Nonempty ι]
  (f : ι → ι' → α) (Ha : ∀ j, BddAbove (Set.range (f · j))) (Hb : ∀ i, BddBelow (Set.range (f i
    ))) :
    ⨆ i, ⨅ j, f i j ≤ ⨅ j, ⨆ i, f i j
```

theorem range_max [source]

```
theorem BddAbove.range_max (hf : BddAbove (Set.range f)) (hg : BddAbove (Set.range g)) :
    BddAbove (Set.range (max f g))
```

theorem range_min [source]

```
theorem BddBelow.range_min (hf : BddBelow (Set.range f)) (hg : BddBelow (Set.range g)) :
    BddBelow (Set.range (min f g))
```

theorem ciInf_eq_min_cInf_inter_diff [source]

```
theorem ciInf_eq_min_cInf_inter_diff (S T : Set ι)
  [Nonempty (S ∩ T : Set ι)] [Nonempty (S \ T : Set ι)] (hf : BddBelow (f '' S)) :
    ⨅ i : S, f i = (⨅ i : (S ∩ T : Set ι), f i) ⊓ ⨅ i : (S \ T : Set ι), f i
```

theorem lt_ciInf_iff [source]

```
theorem lt_ciInf_iff (hf : BddBelow (Set.range f)) :
    a < iInf f ↔ ∃ b, a < b ∧ ∀ (i : ι), b ≤ f i
```

theorem ciSup_sup_eq [source]

```
theorem ciSup_sup_eq (hf : BddAbove (Set.range f)) (hg : BddAbove (Set.range g)) : ⨆ x, f x ⊔
    g x = (⨆ x, f x) ⊔ ⨆ x, g x
```

theorem ciInf_inf_eq [source]

```
theorem ciInf_inf_eq (hf : BddBelow (Set.range f)) (hg : BddBelow (Set.range g)) : ⨅ x, f x
    ⊓ g x = (⨅ x, f x) ⊓ ⨅ x, g x
```

theorem sup_ciSup [source]

```
theorem sup_ciSup (hf : BddAbove (Set.range f)) : a ⊔ ⨆ x, f x = ⨆ x, a ⊔ f x
```

theorem inf_ciInf [source]

```
theorem inf_ciInf (hf : BddBelow (Set.range f)) : a ⊓ ⨅ x, f x = ⨅ x, a ⊓ f x
```

theorem ciInf_sup_ciInf_le [source]

```
theorem ciInf_sup_ciInf_le (hf : BddBelow (Set.range f)) (hg : BddBelow (Set.range g)) :
    (⨅ i, f i) ⊔ ⨅ i, g i ≤ ⨅ i, f i ⊔ g i
```

theorem le_ciSup_inf_ciSup [source]

```
theorem le_ciSup_inf_ciSup (hf : BddAbove (Set.range f)) (hg : BddAbove (Set.range g)) :
    ⨆ (i : ι), f i ⊓ g i ≤ (⨆ (i : ι), f i) ⊓ ⨆ (i : ι), g i
```

theorem subset [source]

```
theorem QuasiconvexOn.subset
  {𝕜 E β : Type*} [Semiring 𝕜] [PartialOrder 𝕜] [AddCommMonoid E] [LE β] [SMul 𝕜 E]
  {s : Set E} {f : E → β} (h : QuasiconvexOn 𝕜 s f) {t : Set E} (hts : t ⊆ s) (ht : Convex 𝕜 t)
    :
    QuasiconvexOn 𝕜 t f
```

theorem mem_segment_le_max [source]

```
theorem QuasiconvexOn.mem_segment_le_max
  {𝕜 E β : Type*} [Semiring 𝕜] [PartialOrder 𝕜] [AddCommMonoid E] [SemilatticeSup β] [SMul 𝕜 E
    ]
  {s : Set E} {f : E → β} (h : QuasiconvexOn 𝕜 s f)
  {x y z : E} (hx : x ∈ s) (hy : y ∈ s) (hz : z ∈ segment 𝕜 x y):
    f z ≤ f x ⊔ f y
```

theorem min_le_mem_segment [source]

```
theorem QuasiconcaveOn.min_le_mem_segment
  {𝕜 E β : Type*} [Semiring 𝕜] [PartialOrder 𝕜] [AddCommMonoid E] [SemilatticeInf β] [SMul 𝕜 E
    ]
  {s : Set E} {f : E → β} (h : QuasiconcaveOn 𝕜 s f)
  {x y z : E} (hx : x ∈ s) (hy : y ∈ s) (hz : z ∈ segment 𝕜 x y):
    f x ⊓ f y ≤ f z
```

theorem bddBelow [source]

```
theorem LowerSemicontinuousOn.bddBelow {α : Type*} [TopologicalSpace α] {S : Set α} {g : α → ℝ}
    (hg : LowerSemicontinuousOn g S) (hS : IsCompact S) : BddBelow (g '' S)
```

theorem max [source]

```
theorem LowerSemicontinuousOn.max {α : Type*} [TopologicalSpace α] {S : Set α} {f g : α → ℝ}
    (hf : LowerSemicontinuousOn f S) (hg : LowerSemicontinuousOn g S) :
    LowerSemicontinuousOn (fun x ↦ max (f x) (g x)) S
```

theorem lowerSemicontinuousOn_iff_isClosed_preimage [source]

```
theorem lowerSemicontinuousOn_iff_isClosed_preimage {f : α → γ} [IsClosed s] :
    LowerSemicontinuousOn f s ↔ ∀ y, IsClosed (s ∩ f ⁻¹' Set.Iic y)
```

theorem isConnected [source]

```
theorem segment.isConnected {E : Type u_1} [AddCommGroup E] [Module ℝ E] [TopologicalSpace E] [
    ContinuousAdd E] [ContinuousSMul ℝ E] (a b : E) :
    IsConnected (segment ℝ a b)
```

theorem range_inf_of_image2 [source]

```
theorem BddAbove.range_inf_of_image2 {M N α : Type*} {f : M → N → α} [
    ConditionallyCompleteLinearOrder α]
  {S : Set M} {T : Set N} (h_bddA : BddAbove (Set.image2 f S T)) (h_bddB : BddBelow (Set.image2
     f S T)) :
    BddAbove (Set.range fun y : T ↦ ⨅ x : S, f x y)
```

theorem range_sup_of_image2 [source]

```
theorem BddBelow.range_sup_of_image2 {M N α : Type*} {f : M → N → α} [
    ConditionallyCompleteLinearOrder α]
  {S : Set M} {T : Set N} (h_bddA : BddAbove (Set.image2 f S T)) (h_bddB : BddBelow (Set.image2
     f S T)) :
     BddBelow (Set.range fun y : T ↦ ⨆ x : S, f x y)
```

theorem ciInf_le_ciInf_of_subset [source]

```
theorem ciInf_le_ciInf_of_subset {α β : Type*} [ConditionallyCompleteLattice α]
  {f : β → α} {s t : Set β} (hs : s.Nonempty) (hf : BddBelow (f '' t)) (hst : s ⊆ t) :
    ⨅ x : t, f x ≤ ⨅ x : s, f x
```

theorem dite_top [source]

```
theorem LowerSemicontinuousOn.dite_top {α β : Type*} [TopologicalSpace α] [Preorder β] [
    OrderTop β]
  {s : Set α} (p : α → Prop) [DecidablePred p] {f : (a : α) → p a → β}
  (hf : LowerSemicontinuousOn (fun x : Subtype p ↦ f x.val x.prop) {x | x.val ∈ s})
  (h_relatively_closed : ∃ U : Set α, IsClosed U ∧ s ∩ U = s ∩ setOf p) :
    LowerSemicontinuousOn (fun x ↦ dite (p x) (f x) (fun _ ↦ ⊤)) s
```

theorem LowerSemicontinuousOn.comp_continuousOn [source]

```
theorem LowerSemicontinuousOn.comp_continuousOn {α β γ : Type*}
  [TopologicalSpace α] [TopologicalSpace β] [Preorder γ] {f : α → β} {s : Set α} {g : β → γ} {t
     : Set β}
  (hg : LowerSemicontinuousOn g t) (hf : ContinuousOn f s) (h : Set.MapsTo f s t) :
    LowerSemicontinuousOn (g ∘ f) s
```

theorem UpperSemicontinuousOn.comp_continuousOn [source]

```
theorem UpperSemicontinuousOn.comp_continuousOn {α β γ : Type*}
  [TopologicalSpace α] [TopologicalSpace β] [Preorder γ] {f : α → β} {s : Set α} {g : β → γ} {t
     : Set β}
  (hg : UpperSemicontinuousOn g t) (hf : ContinuousOn f s) (h : Set.MapsTo f s t) :
    UpperSemicontinuousOn (g ∘ f) s
```

theorem ite_top [source]

```
theorem LowerSemicontinuousOn.ite_top {α β : Type*} [TopologicalSpace α] [Preorder β] [OrderTop
    β]
  {s : Set α} (p : α → Prop) [DecidablePred p] {f : (a : α) → β} (hf : LowerSemicontinuousOn f
    (s ∩ setOf p))
  (h_relatively_closed : ∃ U : Set α, IsClosed U ∧ s ∩ U = s ∩ setOf p) :
    LowerSemicontinuousOn (fun x ↦ ite (p x) (f x) ⊤) s
```

theorem comp_lowerSemicontinuousOn_strong_assumptions [source]

```
theorem LeftOrdContinuous.comp_lowerSemicontinuousOn_strong_assumptions {α γ δ : Type*}
  [TopologicalSpace α] [LinearOrder γ] [LinearOrder δ] [TopologicalSpace δ] [OrderTopology δ]
  [TopologicalSpace γ] [OrderTopology γ] [DenselyOrdered γ] [DenselyOrdered δ]
  {s : Set α} {g : γ → δ} {f : α → γ} (hg : LeftOrdContinuous g) (hf : LowerSemicontinuousOn f
    s) (hg2 : Monotone g) :
    LowerSemicontinuousOn (g ∘ f) s
```

theorem frequently_lt_of_tendsto [source]

```
theorem UpperSemicontinuousOn.frequently_lt_of_tendsto {α β γ : Type*} [TopologicalSpace β] [
    Preorder γ]
  {f : β → γ} {T : Set β} (hf : UpperSemicontinuousOn f T) {c : γ} {zs : α → β} {z : β}
  {l : Filter α} [l.NeBot] (hzs : l.Tendsto zs (nhds z)) (hx₂ : f z < c) (hzI : ∀ a, zs a ∈ T)
    (hzT : z ∈ T) :
    ∀ᶠ a in l, f (zs a) < c
```

theorem ciInf_insert [source]

```
theorem Finset.ciInf_insert {α β : Type*} [DecidableEq α] [ConditionallyCompleteLattice β]
  (t : Finset α) (ht : t.Nonempty) (x : α) (f : α → β) :
    ⨅ (a : (insert x t : _)), f a = f x ⊓ ⨅ (a : t), f a
```

theorem ciSup_insert [source]

```
theorem Finset.ciSup_insert {α β : Type*} [DecidableEq α] [ConditionallyCompleteLattice β]
  (t : Finset α) (ht : t.Nonempty) (x : α) (f : α → β) :
    ⨆ (a : (insert x t : _)), f a = f x ⊔ ⨆ (a : t), f a
```

theorem sion_exists_min_lowerSemi [source]

```
private theorem sion_exists_min_lowerSemi (a : ℝ) (hc : ∀ y₀ : T, ⨅ (x : S), f (↑x) y₀ ≤ a) (
    z : N) (hzT : z ∈ T) :
    ∃ x ∈ S, f x z ≤ a
```

lemma sion_exists_min_2 [source]

```
private lemma sion_exists_min_2 (y₁ y₂ : N) (hy₁ : y₁ ∈ T) (hy₂ : y₂ ∈ T)
    (a : ℝ) (ha : a < ⨅ x : S, (max (f x y₁) (f x y₂)))
    : ∃ y₀ : T, a < ⨅ x : S, f x y₀
```

lemma sion_exists_min_fin [source]

```
private lemma sion_exists_min_fin
  (h_bddA : BddAbove (Set.image2 f S T)) (h_bddB : BddBelow (Set.image2 f S T))
  (ys : Finset N) (hys_n : ys.Nonempty) (hys : (ys : Set N) ⊆ T)
  (a : ℝ) (ha : a < ⨅ x : S, ⨆ yi : ys, f x yi)
    : ∃ y₀ : T, a < ⨅ x : S, f x y₀
```

theorem sion_minimax [source]

***Sion's Minimax theorem**. Because of `ciSup` and `ciInf` junk values when f isn't bounded, we need to assume that it's bounded above and below.*

```
theorem sion_minimax
  (h_bddA : BddAbove (Set.image2 f S T))
  (h_bddB : BddBelow (Set.image2 f S T))
    : ⨅ x : S, ⨆ y : T, f x y = ⨆ y : T, ⨅ x : S, f x y
```

2.80 QuantumInfo.ForMathlib.Superadditive

QuantumInfo/ForMathlib/Superadditive.lean on GitHub.

def Superadditive [source]

```
def Superadditive (u : ℕ → ℝ) : Prop
```

theorem to_Subadditive [source]

```
theorem to_Subadditive : Subadditive (-u ·)
```

def lim [source]

```
noncomputable def lim (_h : Superadditive u)
```

theorem tendsto_lim [source]

Fekete's lemma for superadditive sequences: a superadditive sequence which is bounded above converges.

```
theorem tendsto_lim (hbdd : BddAbove (Set.range fun n => u n / n)) :
    Filter.Tendsto (fun n => u n / n) .atTop (nhds (h.lim))
```

theorem to_Superadditive [source]

```
theorem to_Superadditive : Superadditive (-u ·)
```

2.81 QuantumInfo.ForMathlib.ULift

QuantumInfo/ForMathlib/ULift.lean on GitHub.

instance ulift [source]

```
instance AddZero.ulift {α : Type u} [AddZero α] : AddZero (ULift.{v, u} α) where
```

instance instStar [source]

```
instance ULift.instStar {𝕜 : Type u} [Star 𝕜] : Star (ULift.{v,u} 𝕜) where
```

theorem star_eq [source]

```
@[simp]
theorem ULift.star_eq {𝕜 : Type u} [Star 𝕜] (x : ULift.{v,u} 𝕜) : star x = .up (star x.down)
```

instance instInvolutiveStar [source]

```
instance ULift.instInvolutiveStar {𝕜 : Type u} [InvolutiveStar 𝕜] : InvolutiveStar (ULift.{v,u}
    𝕜) where
```

instance instStarMul [source]

```
instance ULift.instStarMul {𝕜 : Type u} [Mul 𝕜] [StarMul 𝕜] : StarMul (ULift.{v,u} 𝕜) where
```

instance StarRing (ULift.{v,u} 𝕜) [source]

```
instance {𝕜 : Type u} [NonUnitalNonAssocSemiring 𝕜] [StarRing 𝕜] : StarRing (ULift.{v,u} 𝕜)
    where
```

theorem starRingEnd_down [source]

```
@[simp]
theorem ULift.starRingEnd_down {𝕜 : Type u} (x : ULift.{v,u} 𝕜) [CommSemiring 𝕜] [StarRing 𝕜] :

    ((starRingEnd (ULift.{v, u} 𝕜)) x).down = star x.down
```

instance NormedField (ULift.{v,u} 𝕜) [source]

```
instance {𝕜 : Type u} [NormedField 𝕜] : NormedField (ULift.{v,u} 𝕜) where
```

instance DenselyNormedField (ULift.{v,u} 𝕜) [source]

```
instance {𝕜 : Type u} [DenselyNormedField 𝕜] : DenselyNormedField (ULift.{v,u} 𝕜) where
```

theorem ulift_apply [source]

```
@[simp]
theorem AddEquiv.ulift_apply {α : Type u} [Add α] (x : ULift.{v, u} α) :
    AddEquiv.ulift.{u, v} x = x.down
```

instance RCLike (ULift.{v,u} 𝕜) [source]

```
noncomputable instance {𝕜 : Type u} [RCLike 𝕜] : RCLike (ULift.{v,u} 𝕜) where
```

2.82 QuantumInfo.ForMathlib.Unitary

QuantumInfo/ForMathlib/Unitary.lean on GitHub.

theorem unitary_star_apply_eq [source]

```
@[simp]
theorem unitary_star_apply_eq (U : unitary (E →ₗ[𝕜] E)) (v : E) :
    (star U.val) (U.val v) = v
```

theorem unitary_apply_star_eq [source]

```
@[simp]
theorem unitary_apply_star_eq (U : unitary (E →ₗ[𝕜] E)) (v : E) :
    U.val ((star U.val) v) = v
```

def conj_unitary_eigenspace_equiv [source]

Conjugating a linear map by a unitary operator gives a map whose μ-eigenspace is isomorphic (same dimension) as those of the original linear map.

```
noncomputable def conj_unitary_eigenspace_equiv (T : E →ₗ[𝕜] E) (U : unitary (E →ₗ[𝕜] E)) (μ :
    𝕜) :
    eigenspace T μ ≃ₗ[𝕜] eigenspace (U.val * T * star (U.val)) μ where
```

theorem conj_unitary_IsSymmetric [source]

A symmetric operator conjugated by a unitary is symmetric.

```
theorem conj_unitary_IsSymmetric (U : unitary (E →ₗ[𝕜] E)) (hT : T.IsSymmetric) :
    (U.val * T * star U.val).IsSymmetric
```

def conj_unitary_eigenvalue_equiv [source]

There is an equivalence between the eigenvalues of a finite dimensional symmetric operator, and the eigenvalues of that operator conjugated by a unitary.

```
def conj_unitary_eigenvalue_equiv (U : unitary (E →ₗ[𝕜] E)) (hT : T.IsSymmetric) :
    { σ : Equiv.Perm (Fin n) // (hT.conj_unitary_IsSymmetric U).eigenvalues hn = hT.eigenvalues
     hn ∘ σ }
```

2.83 QuantumInfo.Regularized

QuantumInfo/Regularized.lean on GitHub.

def InfRegularized [source]

An `InfRegularized` value is the lim sup of value at each natural number, but requires a proof of lower- and upper-bounds to be defined.

```
noncomputable def InfRegularized (fn : ℕ → T) {lb ub : T}
    (_ : ∀ n, lb ≤ fn n) (_ : ∀ n, fn n ≤ ub) : T
```

def SupRegularized [source]

A `SupRegularized` value is the lim sup of value at each natural number, but requires a proof of lower- and upper-bounds to be defined.

```
noncomputable def SupRegularized (fn : ℕ → T) {lb ub : T}
    (_ : ∀ n, lb ≤ fn n) (_ : ∀ n, fn n ≤ ub) : T
```

theorem InfRegularized.lb [source]

The `InfRegularized` value is also lower bounded.

```
theorem lb : _lb ≤ InfRegularized fn hl hu
```

theorem InfRegularized.ub [source]

The `InfRegularized` value is also upper bounded.

```
theorem ub : InfRegularized fn hl hu ≤ _ub
```

theorem anti_inf [source]

For `Antitone` functions, the `InfRegularized` is the supremum of values.

```
theorem anti_inf (h : Antitone fn) :
    InfRegularized fn hl hu = sInf (Set.range fn)
```

theorem anti_ub [source]

For `Antitone` functions, the `InfRegularized` is lower bounded by any particular value.

```
theorem anti_ub (h : Antitone fn) : ∀ n, InfRegularized fn hl hu ≤ fn n
```

theorem SupRegularized.lb [source]

The `SupRegularized` value is also lower bounded.

```
theorem lb : _lb ≤ SupRegularized fn hl hu
```

theorem SupRegularized.ub [source]

The `SupRegularized` value is also upper bounded.

```
theorem ub : SupRegularized fn hl hu ≤ _ub
```

theorem mono_sup [source]

For `Monotone` functions, the `SupRegularized` is the supremum of values.

```
theorem mono_sup (h : Monotone fn) :
    SupRegularized fn hl hu = sSup { fn n | n : ℕ}
```

theorem mono_lb [source]

For `Monotone` functions, the `SupRegularized` is lower bounded by any particular value.

```
theorem mono_lb (h : Monotone fn) : ∀ n, fn n ≤ SupRegularized fn hl hu
```

theorem to_SupRegularized [source]

```
theorem InfRegularized.to_SupRegularized : InfRegularized fn hl hu = -SupRegularized (-fn ·)
    (lb := -_ub) (ub := -_lb) (neg_le_neg_iff.mpr <| hu ·) (neg_le_neg_iff.mpr <| hl ·)
```

theorem to_InfRegularized [source]

```
theorem SupRegularized.to_InfRegularized : SupRegularized fn hl hu = -InfRegularized (-fn ·)
    (lb := -_ub) (ub := -_lb) (neg_le_neg_iff.mpr <| hu ·) (neg_le_neg_iff.mpr <| hl ·)
```

theorem anti_tendsto [source]

For `Antitone` functions, the value `Filter.Tendsto` the `InfRegularized` value.

```
theorem InfRegularized.anti_tendsto (h : Antitone fn) :
    Filter.Tendsto fn .atTop (nhds (InfRegularized fn hl hu))
```

theorem of_Subadditive [source]

```
theorem InfRegularized.of_Subadditive (hf : Subadditive (fun n ↦ fn n * n))
    :
    hf.lim = InfRegularized fn hl hu
```

2.84 QuantumInfo.StatMech.Hamiltonian

QuantumInfo/StatMech/Hamiltonian.lean on GitHub.

structure MicroHamiltonian [source]

The Hamiltonian for a microcanonical ensemble, parameterized by any extrinsic parameters of data D. Since it's an arbitrary type, 'T' would be nice, but we use 'D' for data to avoid confusion with temperature. We define Hamiltonians as an energy function on a real manifold, whose dimension n possibly depends on the data D. Energy is a WithTop ℝ, ⊤ is used to mean "excluded as a possibility in phase space". This formalization does exclude the possibility of discrete degrees of freedom; it is not __completely__ general. A better formalization could have an arbitrary measurable space.

```
structure MicroHamiltonian (D : Type) where
  dim : D → Type
  [dim_fin : ∀ d, Fintype (dim d)]
  H : {d : D} → (dim d → ℝ) → WithTop ℝ
```

instance microHamiltonianFintype [source]

We add the dim_fin to the instance cache so that things like the measure can be synthesized

```
instance microHamiltonianFintype (H : MicroHamiltonian D) (d : D) : Fintype (H.dim d)
```

abbrev NVEHamiltonian [source]

The standard microcanonical ensemble Hamiltonian, where the data is the particle number N and the volume V.

```
abbrev NVEHamiltonian
```

def N [source]

Helper to get the number in an N-V Hamiltonian

```
def NVEHamiltonian.N : (ℕ × ℝ) → ℕ
```

def V [source]

Helper to get the volume in an N-V Hamiltonian

```
def NVEHamiltonian.V : (ℕ × ℝ) → ℝ
```

2.85 QuantumInfo.StatMech.IdealGas

QuantumInfo/StatMech/IdealGas.lean on GitHub.

def IdealGas [source]

The Hamiltonian for an ideal gas: particles live in a cube of volume V^(1/3), and each contributes an energy p^2/2. The per-particle mass is normalized to 1.

```
def IdealGas : NVEHamiltonian where
```

theorem PartitionZ_eq [source]

The partition function Z for an ideal gas.

```
theorem PartitionZ_eq (hV : 0 < V) (hβ : 0 < β) :
    IdealGas.PartitionZ (n,V) β = V^n * (2 * Real.pi / β)^(3 * n / 2 : ℝ)
```

theorem HelmholtzA_eq [source]

The Helmholtz Free Energy A for an ideal gas.

```
theorem HelmholtzA_eq (hV : 0 < V) (hT : 0 < T) : IdealGas.HelmholtzA (n,V) T =
    -n * T * (Real.log V + (3/2) * Real.log (2 * Real.pi * T))
```

theorem ZIntegrable [source]

```
theorem ZIntegrable (hV : 0 < V) (hβ : 0 < β) : IdealGas.ZIntegrable (n,V) β
```

theorem IdealGasLaw [source]

The ideal gas law: $PV = nRT$. In our unitsless system, $R = 1$.

```
theorem IdealGasLaw (hV : 0 < V) (hT : 0 < T) :
    let P
```

2.86 QuantumInfo.StatMech.ThermoQuantities

QuantumInfo/StatMech/ThermoQuantities.lean on GitHub.

def PartitionZ [source]

The partition function corresponding to a given MicroHamiltonian. This is a function taking a thermodynamic β, not a temperature. It also depends on the data D defining the system extrinsincs.

** Ideally this would be an NNReal, but $\int$ (NNReal) doesn't work right now, so it would just be a separate proof anyway*

```
def PartitionZ (β : ℝ) : ℝ
```

def PartitionZT [source]

The partition function as a function of temperature T instead of β.

```
def PartitionZT (T : ℝ) : ℝ
```

def InternalU [source]

The Internal Energy, U or E, defined as -∂(ln Z)/∂β. Parameterized here with β.

```
def InternalU (β : ℝ) : ℝ
```

def HelmholtzA [source]

*The Helmholtz Free Energy, -T * ln Z. Also denoted F. Parameterized here with temperature T, not β.*

```
def HelmholtzA (T : ℝ) : ℝ
```

def EntropyS [source]

The entropy, defined as the -∂A/∂T. Function of T.

```
def EntropyS (T : ℝ) : ℝ
```

def EntropySβ [source]

*The entropy, defined as ln Z + β*U. Function of β.*

```
def EntropySβ (β : ℝ) : ℝ
```

def ZIntegrable [source]

To be able to compute or define anything from a Hamiltonian, we need its partition function to be a computable integral. A Hamiltonian is ZIntegrable at β if PartitionZ is Lesbegue integrable and nonzero.

```
def ZIntegrable (β : ℝ) : Prop
```

def PositiveβIntegrable [source]

This Prop defines the most common case of ZIntegrable, that it is integrable at all finite temperatures (aka all positive β).

```
def PositiveβIntegrable : Prop
```

theorem DifferentiableAt_Z_if_ZIntegrable [source]

```
theorem DifferentiableAt_Z_if_ZIntegrable {β : ℝ} (h : H.ZIntegrable d β) : ContDiffAt ℝ ω (H.
    PartitionZ d) β
```

theorem entropy_A_eq_entropy_Z [source]

The two definitions of entropy, in terms of T or β, are equivalent.

```
theorem entropy_A_eq_entropy_Z (T β : ℝ) (hβT : T * β = 1) (hi : H.ZIntegrable d β)
    : EntropyS H d T = EntropySβ H d β
```

theorem β_eq_deriv_S_U [source]

The "definition of temperature from entropy": $1/T = (\partial S/\partial U)$, when the derivative is at constant extrinsic d (typically N/V). Here we use β instead of $1/T$ on the left, and express the right actually as $(\partial S/\partial\beta)/(\partial U/\partial\beta)$, as all our things are ultimately parameterized by β.

```
theorem β_eq_deriv_S_U {β : ℝ} (hi : H.ZIntegrable d β) : β = (deriv (H.EntropySβ d) β) / deriv
    (H.InternalU d) β
```

def Pressure [source]

Pressure, as a function of T. Defined as the conjugate variable to volume.

```
def Pressure (T : ℝ) : ℝ
```