Physlib Monthly Report

1–31 July 2026

Contributors: Aleksei Bykov, Alex Zughaid, Andrea Pari, August, Augustinas Malinauskas, doxtor6, Florian Wiesner, Giuseppe Barbalinardo, giuseppe.sorge, Gregory Loges, Jake Uribe, Joseph Tooby-Smith, juanjfndz, Kenneth Jusino, KrystianNowakowski, Lazar Milikic, nateabr, Nicola Bernini, Nicolas Rouquette, Philip Haynes, Robby Sneiderman, Robert Joseph, Shlok Vaibhav Singh, Tracy McSheery, Vasily Ilin

Reviewers: Daniel Morrison, doxtor6, Gregory Loges, Joseph Tooby-Smith, nateabr, Zhi Kai Pong

Generated 14 August 2026 from commit `471cf7f`.

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 July 2026 on branch `master` of `leanprover-community/physlib`, between commits `05930fe` and `471cf7f`. Over that period 25 contributors landed 108 commits, changing 36,698 lines across 356 files and adding 962 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 . . . 3
- 1.4 New declarations by kind . . . 3

2 Details by file **4**
- 2.1 `Physlib.ClassicalMechanics.DampedHarmonicOscillator.Basic` . . . 4
- 2.2 `Physlib.ClassicalMechanics.DampedHarmonicOscillator.Solution` . . . 6
- 2.3 `Physlib.ClassicalMechanics.HarmonicOscillator.Geometric.Basic` . . . 7
- 2.4 `Physlib.ClassicalMechanics.HarmonicOscillator.Geometric.KineticEnergy` 8
- 2.5 `Physlib.ClassicalMechanics.HarmonicOscillator.Geometric.Trajectory` . . 11
- 2.6 `Physlib.ClassicalMechanics.Lagrangian.TotalDerivativeEquivalence` . . . 12
- 2.7 `Physlib.ClassicalMechanics.RigidBody.AngularMomentum` . . . 13
- 2.8 `Physlib.ClassicalMechanics.RigidBody.AngularVelocity` . . . 13
- 2.9 `Physlib.ClassicalMechanics.RigidBody.Basic` . . . 17
- 2.10 `Physlib.ClassicalMechanics.RigidBody.KineticEnergy` . . . 18
- 2.11 `Physlib.ClassicalMechanics.RigidBody.Motion` . . . 20
- 2.12 `Physlib.CondensedMatter.Thermoelectric.Basic` . . . 24
- 2.13 `Physlib.FluidDynamics.CauchyFlow.Basic` . . . 26
- 2.14 `Physlib.FluidDynamics.CauchyFlow.BodyForce` . . . 26

2.15 Physlib.FluidDynamics.CauchyFlow.Inviscid . . . 26
2.16 Physlib.FluidDynamics.CauchyFlow.Momentum . . . 27
2.17 Physlib.FluidDynamics.CauchyFlow.NavierStokes . . . 28
2.18 Physlib.FluidDynamics.CauchyFlow.Newtonian . . . 28
2.19 Physlib.FluidDynamics.Euler.Basic . . . 28
2.20 Physlib.FluidDynamics.FluidFlow.Basic . . . 29
2.21 Physlib.FluidDynamics.FluidFlow.Continuity . . . 29
2.22 Physlib.FluidDynamics.FluidFlow.Incompressible . . . 30
2.23 Physlib.FluidDynamics.FluidFlow.Kinematics . . . 31
2.24 Physlib.FluidDynamics.FluidFlow.Momentum . . . 31
2.25 Physlib.FluidDynamics.FluidFlow.Newtonian . . . 32
2.26 Physlib.FluidDynamics.ThermodynamicCauchyFlow.Basic . . . 33
2.27 Physlib.FluidDynamics.ThermodynamicCauchyFlow.Bernoulli . . . 33
2.28 Physlib.FluidDynamics.ThermodynamicCauchyFlow.Isentropic . . . 34
2.29 Physlib.Mathematics.CrossProduct . . . 34
2.30 Physlib.Mathematics.CrossProductMatrix . . . 34
2.31 Physlib.Mathematics.Distribution.Basic . . . 35
2.32 Physlib.Mathematics.InnerProductSpace.Gaussian . . . 36
2.33 Physlib.Mathematics.KroneckerDelta.Basic . . . 37
2.34 Physlib.Mathematics.KroneckerDelta.Contraction . . . 37
2.35 Physlib.Mathematics.OrthogonalMatrix . . . 40
2.36 Physlib.Mathematics.Resolvent . . . 40
2.37 Physlib.Mathematics.SpecialFunctions.PhysHermite . . . 41
2.38 Physlib.Particles.StandardModel.Basic . . . 47
2.39 Physlib.Particles.StandardModel.Fermions.DownSinglet . . . 48
2.40 Physlib.Particles.StandardModel.Fermions.LeptonDoublet . . . 50
2.41 Physlib.Particles.StandardModel.Fermions.LeptonSinglet . . . 53
2.42 Physlib.Particles.StandardModel.Fermions.QuarkDoublet . . . 55
2.43 Physlib.Particles.StandardModel.Fermions.UpSinglet . . . 57
2.44 Physlib.QuantumMechanics.FreeParticle.Basic . . . 60
2.45 Physlib.QuantumMechanics.HarmonicOscillator.Basic . . . 61
2.46 Physlib.QuantumMechanics.HarmonicOscillator.OneDimension.Basic . . . 64
2.47 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.Basic . . . 65
2.48 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.DirichletSubmodule . . . 69
2.49 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.Fourier . . . 70
2.50 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.PolyBddSchwartzSubmodule 71
2.51 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.SchwartzSubmodule . . . 71
2.52 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.SobolevSubmodule . . . 72
2.53 Physlib.QuantumMechanics.InfiniteSquareWell.Basic . . . 73
2.54 Physlib.QuantumMechanics.Operators.Multiplication . . . 74
2.55 Physlib.QuantumMechanics.Operators.OneDimension.Momentum . . . 77
2.56 Physlib.QuantumMechanics.Operators.OneDimension.Parity . . . 78
2.57 Physlib.QuantumMechanics.Operators.OneDimension.Position . . . 78
2.58 Physlib.QuantumMechanics.Operators.OneDimension.Unbounded . . . 78
2.59 Physlib.QuantumMechanics.Operators.SpectralTheory.SelfAdjoint . . . 78
2.60 Physlib.QuantumMechanics.Operators.SpectralTheory.Symmetric . . . 79
2.61 Physlib.QuantumMechanics.Operators.Unbounded . . . 79
2.62 Physlib.QuantumMechanics.QuantumSystem.Basic . . . 81
2.63 Physlib.QuantumMechanics.RectangularBarrier.Basic . . . 83
2.64 Physlib.Relativity.Fermions.Dirac.Basic . . . 85
2.65 Physlib.Relativity.Fermions.Weyl.DualLeftHanded . . . 86

2.66 Physlib.Relativity.Fermions.Weyl.DualRightHanded . . . 87
2.67 Physlib.Relativity.Fermions.Weyl.Duals . . . 89
2.68 Physlib.Relativity.Fermions.Weyl.LeftHanded . . . 90
2.69 Physlib.Relativity.Fermions.Weyl.RightHanded . . . 91
2.70 Physlib.Relativity.LorentzGroup.Basic . . . 92
2.71 Physlib.Relativity.LorentzGroup.Restricted.FromBoostRotation . . . 93
2.72 Physlib.Relativity.Tensors.Basic . . . 93
2.73 Physlib.Relativity.Tensors.ComplexTensor.Metrics.Basic . . . 94
2.74 Physlib.Relativity.Tensors.ComplexTensor.Units.Basic . . . 94
2.75 Physlib.Relativity.Tensors.Conjugation.Basic . . . 95
2.76 Physlib.Relativity.Tensors.Contraction.CrossToEnd . . . 95
2.77 Physlib.Relativity.Tensors.Contraction.CrossToSlot . . . 98
2.78 Physlib.Relativity.Tensors.Contraction.Products . . . 99
2.79 Physlib.Relativity.Tensors.Contraction.Pure . . . 100
2.80 Physlib.Relativity.Tensors.Contraction.SuccSuccAbove . . . 101
2.81 Physlib.Relativity.Tensors.Contraction.UnitTensorContraction . . . 101
2.82 Physlib.Relativity.Tensors.Dual . . . 104
2.83 Physlib.Relativity.Tensors.Elab . . . 106
2.84 Physlib.Relativity.Tensors.Evaluation . . . 109
2.85 Physlib.Relativity.Tensors.LeviCivita.Basic . . . 110
2.86 Physlib.Relativity.Tensors.LeviCivita.Contractions . . . 110
2.87 Physlib.Relativity.Tensors.RealTensor.Representation.Contraction . . . 111
2.88 Physlib.Relativity.Tensors.Reindexing . . . 112
2.89 Physlib.Relativity.Tensors.TensorSpecies.Basic . . . 113
2.90 Physlib.SpaceAndTime.Time.Derivatives . . . 114
2.91 Physlib.SpaceAndTime.Time.MatrixDerivatives . . . 114
2.92 Physlib.Units.Dimension . . . 115
2.93 Physlib.Units.ISQBridge . . . 117
2.94 Physlib.Units.ISQDimensionBase . . . 119
2.95 Physlib.Units.LTMCTDimensionBase . . . 120
2.96 Physlib.Units.ParametricDimensionExamples . . . 122
2.97 Physlib.Units.ParametricUnits . . . 123
2.98 Physlib.Units.UnitDependent . . . 124
2.99 Physlib.Units.WithDim.Basic . . . 124
2.100 PhyslibAlpha.ClassicalFieldTheory.Local.EulerLagrangeEquation . . . 126
2.101 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.ChargeBalance . . . 127
2.102 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.EffectivePotential 127
2.103 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.GaugeSlice . . . 128
2.104 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.GaugeTorus . . . 129
2.105 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.Invariants . . . 130
2.106 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.Module . . . 138
2.107 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.OrbitRepresentative 139
2.108 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.SwapDoublet . . . 140
2.109 QuantumInfo.ClassicalInfo.Prob . . . 141
2.110 QuantumInfo.Entropy.DPI . . . 141
2.111 QuantumInfo.Entropy.Relative . . . 142
2.112 scripts.autolabel . . . 143

1 Introduction

During Jul 2026, 25 contributors submitted 108 commits that changed 36,698 lines (+23,053 / -13,645) across 356 files spanning 36 top-level areas of the repository. Of those, 93 files were newly added and 8 were removed outright. This report catalogues 962 newly added Lean declarations: 615 lemmas, 214 defs, 58 instances, 20 structures, 18 theorems, 17 abbrevs, 9 informal lemmas, 7 informal definitions, and 4 inductives. We're delighted to recognize the following first-time contributors this month: Alex Zughaid, August, Augustinas Malinauskas, Giuseppe Barbalinardo, Jake Uribe, Kenneth Jusino, KrystianNowakowski, Nicolas Rouquette, Philip Haynes, and Robert Joseph.

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
Aleksei Bykov	179	2
Alex Zughaid	4	1
Andrea Pari	1,495	1
August	560	7
Augustinas Malinauskas	31	1
doxtor6	461	4
Florian Wiesner	1,323	1
Giuseppe Barbalinardo	214	1
giuseppe.sorge	1,590	12
Gregory Loges	3,947	17
Jake Uribe	23	2
Joseph Tooby-Smith	6,131	18
juanjfndz	101	1
Kenneth Jusino	68	2
KrystianNowakowski	24	1
Lazar Milikic	199	3
nateabr	1,322	5
Nicola Bernini	93	3
Nicolas Rouquette	1,669	2
Philip Haynes	221	1
Robby Sneiderman	1,684	11
Robert Joseph	38	1
Shlok Vaibhav Singh	827	4
Tracy McSheery	19	1
Vasily Ilin	15,181	6

1.2 Reviewers

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

Reviewer	PRs Reviewed

Daniel Morrison	14
doxtor6	1
Gregory Loges	11
Joseph Tooby-Smith	87
nateabr	17
Zhi Kai Pong	4

1.3 Modified subfolders

Subfolder	Files	New	Deleted	Additions	Deletions
`Physlib/Relativity`	60	+11	-2	+4,186	-2,287
`Physlib/QuantumMechanics`	57	+16	-5	+3,178	-1,368
`Physlib/QFT`	29	–	–	+1,080	-2,777
`Physlib/Particles`	34	+6	–	+1,952	-1,203
`Physlib/SpaceAndTime`	22	+6	–	+1,148	-1,826
`Physlib/ClassicalMechanics`	22	+10	–	+2,363	-595
`Physlib/Mathematics`	19	+5	–	+1,478	-870
`PhyslibAlpha/Particles`	8	+8	–	+1,736	-0
`scripts`	5	+3	–	+1,687	-6
`Physlib/FluidDynamics`	19	+15	-1	+1,156	-235
`Physlib/Units`	12	+5	–	+967	-326
`Physlib/Electromagnetism`	13	+1	–	+404	-815
`Physlib/StatisticalMechanics`	4	+1	–	+298	-726
`Physlib/StringTheory`	9	–	–	+121	-269
`Physlib/CondensedMatter`	2	+1	–	+249	-100
`QuantumInfo/Entropy`	2	–	–	+201	-15
`.github/workflows`	3	+2	–	+201	-0
`QuantumInfo/ForMathlib`	13	–	–	+104	-65
`Physlib.lean`	1	–	–	+106	-52
`QuantumInfo/Capacity`	1	–	–	+114	-34
`PhyslibAlpha/ClassicalFieldTheory`	1	+1	–	+100	-0
`Physlib/ClassicalFieldTheory`	1	+1	–	+91	-0
`docs`	1	+1	–	+50	-0
`Physlib/Cosmology`	1	–	–	+11	-23
`lake-manifest.json`	1	–	–	+17	-17
`QuantumInfo/Operators`	1	–	–	+11	-10
`QuantumInfo/ClassicalInfo`	4	–	–	+10	-6
`QuantumInfo/States`	3	–	–	+6	-6
`PhyslibAlpha.lean`	1	–	–	+9	-0
`PhyslibAlpha/QuantumMechanics`	1	–	–	+4	-4
`lakefile.toml`	1	–	–	+6	-2
`README.md`	1	–	–	+1	-6
`.gitignore`	1	–	–	+4	-0
`QuantumInfo/Channels`	1	–	–	+1	-1
`lean-toolchain`	1	–	–	+1	-1
`scripts/MetaPrograms`	1	–	–	+2	-0

1.4 New declarations by kind

Kind	Count
lemma	615
def	214
instance	58
structure	20
theorem	18
abbrev	17
informal_lemma	9
informal_definition	7
inductive	4

2 Details by file

2.1 Physlib.ClassicalMechanics.DampedHarmonicOscillator.Basic

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

lemma acceleration_eq_of_equationOfMotion [source]

Solving the equation of motion for the acceleration: along a solution the second derivative is `-(k/m) x - (γ/m) ẋ`.

```
lemma acceleration_eq_of_equationOfMotion (z : Time → EuclideanSpace ℝ (Fin 1))
    (hEOM : S.EquationOfMotion z) (t : Time) :
    ∂ₜ (∂ₜ z) t = (-(S.m⁻¹ * S.k)) • z t + (-(S.m⁻¹ * S.γ)) • ∂ₜ z t
```

lemma isUnderdamped_or_isCriticallyDamped_or_isOverdamped [source]

Every damped oscillator is underdamped, critically damped, or overdamped.

```
lemma isUnderdamped_or_isCriticallyDamped_or_isOverdamped :
    S.IsUnderdamped ∨ S.IsCriticallyDamped ∨ S.IsOverdamped
```

def lagrangian [source]

The Caldirola–Kanai lagrangian of the damped harmonic oscillator. It is the lagrangian of the underlying undamped harmonic oscillator multiplied by the time-dependent factor `exp (γ/m * t)`.

```
noncomputable def lagrangian (t : Time) (x v : EuclideanSpace ℝ (Fin 1)) : ℝ
```

lemma lagrangian_eq [source]

```
lemma lagrangian_eq : S.lagrangian = fun (t : Time) (x v : EuclideanSpace ℝ (Fin 1)) =>
    exp (S.γ / S.m * t) * (1 / (2 : ℝ) * S.m * ⟪v, v⟫_ℝ - 1 / (2 : ℝ) * S.k * ⟪x, x⟫_ℝ)
```

lemma lagrangian_eq_exp_mul_kineticEnergy_sub_potentialEnergy [source]

```
lemma lagrangian_eq_exp_mul_kineticEnergy_sub_potentialEnergy (t : Time)
    (xₜ : Time → EuclideanSpace ℝ (Fin 1)) :
    S.lagrangian t (xₜ t) (∂ₜ xₜ t) =
      exp (S.γ / S.m * t) * (S.kineticEnergy xₜ t - S.potentialEnergy (xₜ t))
```

lemma lagrangian_of_isUndamped [source]

When the damping coefficient is zero, the Caldirola–Kanai lagrangian is the lagrangian of the underlying undamped harmonic oscillator.

```
lemma lagrangian_of_isUndamped (hS : S.IsUndamped) :
    S.lagrangian = S.toHarmonicOscillator.lagrangian
```

lemma contDiff_lagrangian [source]

```
@[fun_prop]
lemma contDiff_lagrangian (n : WithTop ℕ∞) : ContDiff ℝ n ↿S.lagrangian
```

lemma gradient_const_mul [source]

```
private lemma gradient_const_mul {f : EuclideanSpace ℝ (Fin 1) → ℝ} {x : EuclideanSpace ℝ (Fin
    1)}
    (c : ℝ) (hf : DifferentiableAt ℝ f x) :
    gradient (fun y => c * f y) x = c • gradient f x
```

lemma gradient_lagrangian_position_eq [source]

```
lemma gradient_lagrangian_position_eq (t : Time) (x v : EuclideanSpace ℝ (Fin 1)) :
    gradient (fun x => S.lagrangian t x v) x = -(exp (S.γ / S.m * t) * S.k) • x
```

lemma gradient_lagrangian_velocity_eq [source]

```
lemma gradient_lagrangian_velocity_eq (t : Time) (x v : EuclideanSpace ℝ (Fin 1)) :
    gradient (S.lagrangian t x) v = (exp (S.γ / S.m * t) * S.m) • v
```

def gradLagrangian [source]

The variational gradient of the action of the damped harmonic oscillator.

```
noncomputable def gradLagrangian (xₜ : Time → EuclideanSpace ℝ (Fin 1)) :
    Time → EuclideanSpace ℝ (Fin 1)
```

lemma gradLagrangian_eq_eulerLagrangeOp [source]

```
lemma gradLagrangian_eq_eulerLagrangeOp (xₜ : Time → EuclideanSpace ℝ (Fin 1))
    (hq : ContDiff ℝ ∞ xₜ) :
    S.gradLagrangian xₜ = eulerLagrangeOp S.lagrangian xₜ
```

lemma deriv_exp_smul [source]

```
private lemma deriv_exp_smul (a : ℝ) (y : Time → EuclideanSpace ℝ (Fin 1))
    (hy : Differentiable ℝ y) (t : Time) :
    ∂ₜ (fun t' : Time => exp (a * t'.val) • y t') t =
      exp (a * t.val) • (∂ₜ y t + a • y t)
```

lemma gradLagrangian_eq_force [source]

The variational gradient of the Caldirola–Kanai action is the exponential factor times the difference of the force and mass times acceleration appearing in Newton's second law.

```
lemma gradLagrangian_eq_force (xₜ : Time → EuclideanSpace ℝ (Fin 1)) (hx : ContDiff ℝ ∞ xₜ) :
    S.gradLagrangian xₜ = fun t : Time =>
      exp (S.γ / S.m * t) • (force S xₜ t - S.m • ∂ₜ (∂ₜ xₜ) t)
```

lemma equationOfMotion_iff_gradLagrangian_zero [source]

```
lemma equationOfMotion_iff_gradLagrangian_zero (xₜ : Time → EuclideanSpace ℝ (Fin 1))
    (hx : ContDiff ℝ ∞ xₜ) :
    S.EquationOfMotion xₜ ↔ S.gradLagrangian xₜ = 0
```

2.2 Physlib.ClassicalMechanics.DampedHarmonicOscillator.Solution

Physlib/ClassicalMechanics/DampedHarmonicOscillator/Solution.lean on GitHub.

lemma trajectory_contDiff [source]

The selected trajectory is smooth.

```
lemma trajectory_contDiff (IC : InitialConditions) :
    ContDiff ℝ ∞ (S.trajectory IC)
```

lemma trajectory_apply_zero [source]

The selected trajectory has initial position `IC.x₀`.

```
lemma trajectory_apply_zero (IC : InitialConditions) : S.trajectory IC 0 = IC.x₀
```

lemma trajectory_velocity_at_zero [source]

The selected trajectory has initial velocity `IC.v₀`.

```
lemma trajectory_velocity_at_zero (IC : InitialConditions) :
    ∂ₜ (S.trajectory IC) 0 = IC.v₀
```

def phaseVectorField [source]

The phase-space vector field of the damped oscillator, sending `(x, ẋ)` to `(ẋ, -(k/m) x - (γ/m) ẋ)`, as a continuous linear map on `E × E`.

```
private noncomputable def phaseVectorField :
    EuclideanSpace ℝ (Fin 1) × EuclideanSpace ℝ (Fin 1) →L[ℝ]
      EuclideanSpace ℝ (Fin 1) × EuclideanSpace ℝ (Fin 1)
```

lemma phaseVectorField_apply [source]

```
private lemma phaseVectorField_apply (a b : EuclideanSpace ℝ (Fin 1)) :
    S.phaseVectorField (a, b) = (b, (-(S.m⁻¹ * S.k)) • a + (-(S.m⁻¹ * S.γ)) • b)
```

lemma toRealCLE_symm_one [source]

```
private lemma toRealCLE_symm_one : Time.toRealCLE.symm (1 : ℝ) = (1 : Time)
```

lemma hasDerivAt_comp_toRealCLE_symm [source]

Bridge from the time derivative to `HasDerivAt` for a curve reparametrised through the canonical `ℝ ≃L[ℝ] Time` equivalence.

```
private lemma hasDerivAt_comp_toRealCLE_symm (w : Time → EuclideanSpace ℝ (Fin 1)) (τ : ℝ)
    (hw : DifferentiableAt ℝ w (Time.toRealCLE.symm τ)) :
    HasDerivAt (fun τ : ℝ => w (Time.toRealCLE.symm τ))
      (∂ₜ w (Time.toRealCLE.symm τ)) τ
```

lemma phaseCurve_hasDerivAt [source]

The phase curve `τ ↦ (z t, ẋ t)` (with `t = toRealCLE.symm τ`) of a smooth solution `z` solves the first-order phase-space ODE with vector field `phaseVectorField`.

```
private lemma phaseCurve_hasDerivAt (z : Time → EuclideanSpace ℝ (Fin 1))
    (hz : ContDiff ℝ ∞ z) (hEOM : S.EquationOfMotion z) (τ : ℝ) :
    HasDerivAt (fun τ : ℝ => (z (Time.toRealCLE.symm τ), ∂ₜ z (Time.toRealCLE.symm τ)))
      (S.phaseVectorField (z (Time.toRealCLE.symm τ), ∂ₜ z (Time.toRealCLE.symm τ))) τ
```

lemma equationOfMotion_unique [source]

Any two smooth solutions of the damped equation of motion with the same initial position and velocity are equal.

```
lemma equationOfMotion_unique (x y : Time → EuclideanSpace ℝ (Fin 1))
    (hx : ContDiff ℝ ∞ x) (hy : ContDiff ℝ ∞ y)
    (hEOMx : S.EquationOfMotion x) (hEOMy : S.EquationOfMotion y)
    (h0 : x 0 = y 0) (hv0 : ∂ₜ x 0 = ∂ₜ y 0) :
    x = y
```

lemma trajectories_unique [source]

The selected trajectory is the unique smooth solution of the damped equation of motion with the given initial conditions.

```
lemma trajectories_unique (IC : InitialConditions) (x : Time → EuclideanSpace ℝ (Fin 1))
    (hx : ContDiff ℝ ∞ x) (hEOM : S.EquationOfMotion x)
    (hx0 : x 0 = IC.x₀) (hv0 : ∂ₜ x 0 = IC.v₀) :
    x = S.trajectory IC
```

2.3 Physlib.ClassicalMechanics.HarmonicOscillator.Geometric.Basic

Physlib/ClassicalMechanics/HarmonicOscillator/Geometric/Basic.lean on GitHub.

def valDiffeomorph [source]

The global coordinate chart as an analytic diffeomorphism between `ConfigurationSpace` and its `EuclideanSpace ℝ (Fin 1)` model, upgrading `valHomeomorphism` to a smooth identification.

```
def valDiffeomorph :
    ConfigurationSpace ≃ₘ^ω ⟮𝓘(ℝ, EuclideanSpace ℝ (Fin 1)), 𝓘(ℝ, EuclideanSpace ℝ (Fin 1))⟯
      EuclideanSpace ℝ (Fin 1) where
```

instance instNormedAddCommGroupTangent [source]

The model-space normed additive group structure on tangent spaces.

```
instance instNormedAddCommGroupTangent (q : ConfigurationSpace) :
    NormedAddCommGroup (TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q)
```

instance instNormedSpaceTangent [source]

The model-space real normed vector space structure on tangent spaces.

```
instance instNormedSpaceTangent (q : ConfigurationSpace) :
    NormedSpace ℝ (TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q)
```

def tangentCoord [source]

The chart-induced continuous linear equivalence from the tangent space at `q` to the Euclidean coordinate model `EuclideanSpace ℝ (Fin 1)`.

```
def tangentCoord (q : ConfigurationSpace) :
    TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q ≃L[ℝ] EuclideanSpace ℝ (Fin 1) where
```

instance instFiniteDimensionalTangent [source]

Each tangent space is finite-dimensional because it is linearly equivalent to `EuclideanSpace ℝ (Fin 1)` through `tangentCoord`.

```
instance instFiniteDimensionalTangent (q : ConfigurationSpace) :
    FiniteDimensional ℝ (TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q)
```

def toSpace [source]

The position in one-dimensional space associated to the configuration.

```
def toSpace (q : ConfigurationSpace) : Space 1
```

lemma toSpace_apply [source]

The physical-space coordinate associated to a configuration is its chosen global coordinate.

```
lemma toSpace_apply (q : ConfigurationSpace) (i : Fin 1) : q.toSpace i = q.val i
```

2.4 Physlib.ClassicalMechanics.HarmonicOscillator.Geometric.KineticEnergy

Physlib/ClassicalMechanics/HarmonicOscillator/Geometric/KineticEnergy.lean on GitHub.

def massMetricVal [source]

The value of the oscillator mass metric at `q`. It is the mass-scaled pullback of the Euclidean inner product along `tangentCoord q`, so `massMetricVal S q v w = S.m * ⟪tangentCoord q v, tangentCoord q w⟫_ℝ`.

```
noncomputable def massMetricVal (S : HarmonicOscillator) (q : ConfigurationSpace) :
    TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q →L[ℝ]
      TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q →L[ℝ] ℝ
```

lemma massMetricVal_apply [source]

Applying the mass metric value to two tangent vectors gives the mass-scaled Euclidean inner product of their coordinate representatives.

```
lemma massMetricVal_apply
    (S : HarmonicOscillator) (q : ConfigurationSpace)
    (v w : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q) :
    massMetricVal S q v w = S.m * ⟪tangentCoord q v, tangentCoord q w⟫_ℝ
```

lemma tangentCoord_ne_zero [source]

A nonzero tangent vector has nonzero coordinate representative under `tangentCoord`.

```
lemma tangentCoord_ne_zero {q : ConfigurationSpace}
    {v : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q}
    (hv : v ≠ 0) : tangentCoord q v ≠ 0
```

lemma massMetricVal_pos [source]

The oscillator mass metric is positive on nonzero tangent vectors.

```
lemma massMetricVal_pos
    (S : HarmonicOscillator) (q : ConfigurationSpace)
    (v : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q) (hv : v ≠ 0) :
    0 < massMetricVal S q v v
```

lemma massMetricVal_isVonNBounded [source]

The mass metric unit ball is bounded in the model norm.

```
lemma massMetricVal_isVonNBounded
    (S : HarmonicOscillator) (q : ConfigurationSpace) :
    IsVonNBounded ℝ
      {v : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q | massMetricVal S q v v < 1}
```

lemma massMetricVal_contMDiff [source]

The oscillator mass metric is constant in the global tangent-bundle chart.

```
lemma massMetricVal_contMDiff (S : HarmonicOscillator) :
    ContMDiff 𝓘(ℝ, EuclideanSpace ℝ (Fin 1))
      (𝓘(ℝ, EuclideanSpace ℝ (Fin 1)).prod
        𝓘(ℝ, EuclideanSpace ℝ (Fin 1) →L[ℝ]
          EuclideanSpace ℝ (Fin 1) →L[ℝ] ℝ)) ω
      (fun q : ConfigurationSpace =>
```

```
      TotalSpace.mk'
        (EuclideanSpace ℝ (Fin 1) →L[ℝ] EuclideanSpace ℝ (Fin 1) →L[ℝ] ℝ)
        (E := fun q : ConfigurationSpace =>
          TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q →L[ℝ]
            TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q →L[ℝ] ℝ)
        q (massMetricVal S q))
```

def massRiemannianMetric [source]

The mass-scaled Euclidean inner product as a Riemannian metric on the oscillator configuration space.

```
noncomputable def massRiemannianMetric (S : HarmonicOscillator) :
    ContMDiffRiemannianMetric 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) ω
      (EuclideanSpace ℝ (Fin 1))
      (TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) : ConfigurationSpace → Type _) where
```

def geometricKineticEnergy [source]

The defining formula for the oscillator geometric kinetic energy.

```
noncomputable def geometricKineticEnergy
    (S : HarmonicOscillator) (q : ConfigurationSpace)
    (v : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q) : ℝ
```

lemma geometricKineticEnergy_eq [source]

The geometric kinetic energy is one half the mass Riemannian metric applied to `v` twice.

```
lemma geometricKineticEnergy_eq
    (S : HarmonicOscillator) (q : ConfigurationSpace)
    (v : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q) :
    geometricKineticEnergy S q v =
      (1 / 2 : ℝ) * S.massRiemannianMetric.inner q v v
```

lemma massRiemannianMetric_pos [source]

The oscillator mass Riemannian metric is positive definite.

```
lemma massRiemannianMetric_pos
    (S : HarmonicOscillator) (q : ConfigurationSpace)
    (v : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q)
    (hv : v ≠ 0) :
    0 < S.massRiemannianMetric.inner q v v
```

lemma massRiemannianMetric_inner_apply [source]

In the global coordinate, the oscillator mass metric is the mass-scaled Euclidean inner product of coordinate representatives.

```
lemma massRiemannianMetric_inner_apply
    (S : HarmonicOscillator) (q : ConfigurationSpace)
    (v w : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q) :
    S.massRiemannianMetric.inner q v w =
```

```
      S.m * ⟪tangentCoord q v, tangentCoord q w⟫_ℝ
```

lemma geometricKineticEnergy_massMetric_eq [source]

The metric-induced kinetic energy for the mass metric has the standard harmonic-oscillator coordinate form.

```
lemma geometricKineticEnergy_massMetric_eq
    (S : HarmonicOscillator) (q : ConfigurationSpace)
    (v : TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) q) :
    geometricKineticEnergy S q v =
      (1 / 2 : ℝ) * S.m * ⟪tangentCoord q v, tangentCoord q v⟫_ℝ
```

2.5 Physlib.ClassicalMechanics.HarmonicOscillator.Geometric.Trajectory

Physlib/ClassicalMechanics/HarmonicOscillator/Geometric/Trajectory.lean on GitHub.

abbrev Trajectory [source]

A trajectory of the oscillator: a curve in the configuration manifold `Q`.

```
abbrev Trajectory
```

def coord [source]

The coordinate curve of a trajectory: its reading in the global Euclidean coordinate.

```
def coord (γ : Trajectory) : Time → EuclideanSpace ℝ (Fin 1)
```

def toSpace [source]

The physical position of the oscillator along a trajectory, over time.

```
def toSpace (γ : Trajectory) : Time → Space 1
```

lemma coord_apply [source]

```
lemma coord_apply (γ : Trajectory) (t : Time) : coord γ t = (γ t).val
```

lemma contMDiff_iff_contDiff_coord [source]

A trajectory is smooth as a curve in `Q` iff its coordinate curve is ordinarily smooth.

```
lemma contMDiff_iff_contDiff_coord {n : WithTop ℕ∞} (γ : Trajectory) :
    ContMDiff 𝓘(ℝ, Time) 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) n γ ↔
      ContDiff ℝ n (coord γ)
```

def velocity [source]

The geometric velocity of a trajectory at time `t`: a tangent vector at `γ t`.

```
noncomputable def velocity (γ : Trajectory) (t : Time) :
    TangentSpace 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) (γ t)
```

lemma mdifferentiableAt_iff_differentiableAt_coord [source]

A trajectory is `MDifferentiableAt` a time iff its coordinate curve is ordinarily differentiable there, since the two differ by the coordinate diffeomorphism.

```
lemma mdifferentiableAt_iff_differentiableAt_coord (γ : Trajectory) (t : Time) :
    MDifferentiableAt 𝓘(ℝ, Time) 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) γ t ↔
      DifferentiableAt ℝ (coord γ) t
```

lemma mfderiv_eq_mfderiv_coord [source]

A trajectory and its coordinate curve have the same manifold derivative, since they differ only by the coordinate diffeomorphism. This is the one lemma that unfolds the chart; every later result uses only the stable `mfderiv`/`fderiv` API.

```
private lemma mfderiv_eq_mfderiv_coord (γ : Trajectory) (t : Time) :
    mfderiv 𝓘(ℝ, Time) 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) γ t =
      mfderiv 𝓘(ℝ, Time) 𝓘(ℝ, EuclideanSpace ℝ (Fin 1)) (coord γ) t
```

lemma velocity_eq_deriv_coord [source]

In the global coordinate, geometric velocity is represented by the time derivative of the coordinate curve.

```
lemma velocity_eq_deriv_coord (γ : Trajectory) (t : Time) :
    velocity γ t = ∂ₜ (coord γ) t
```

2.6 Physlib.ClassicalMechanics.Lagrangian.TotalDerivativeEquivalence

Physlib/ClassicalMechanics/Lagrangian/TotalDerivativeEquivalence.lean on GitHub.

lemma isTotalTimeDerivative_neg [source]

Elementary fact: if δL is a time derivative, then so is -δL.

```
lemma isTotalTimeDerivative_neg {δL : Time → X → X → ℝ} (h :  IsTotalTimeDerivative δL) :
    IsTotalTimeDerivative (- δL)
```

lemma totalTimeDerivative_contDiff [source]

If δL is a total time derivative (of a smooth function), then it is smooth

```
lemma totalTimeDerivative_contDiff {δL : Time → X → X → ℝ} (h : IsTotalTimeDerivative δL):
    ContDiff ℝ ∞ ↿δL
```

lemma totalTimeDerivative_hasZeroVarGradient [source]

Total time derivative has a variational derivative, which is zero

```
lemma totalTimeDerivative_hasZeroVarGradient [CompleteSpace X] {δL : Time → X → X → ℝ}
    (h : IsTotalTimeDerivative δL) (q : Time → X) (hq : ContDiff ℝ ∞ q):
     HasVarGradientAt (fun q' t => δL t (q' t) (∂ₜ q' t)) (fun _ => 0) q
```

lemma totalTimeDerivative_hasVarGradientAt_equivalence [source]

If two lagrangians, L and L', differ by a total time derivative, and L has a variational derivative grad, then so does L'.

```
lemma totalTimeDerivative_hasVarGradientAt_equivalence [CompleteSpace X] (L δL : Time → X → X
    → ℝ)
    (hδL : IsTotalTimeDerivative δL)
    (q : Time → X)    (hq : ContDiff ℝ ∞ q) (grad : Time → X)
    (hgrad :  HasVarGradientAt (fun q' t => L t (q' t) (fderiv ℝ  q' t 1)) grad q) :
    HasVarGradientAt (fun q' t => (L + δL) t (q' t) (fderiv ℝ q' t 1)) grad q
```

lemma totalTimeDerivative_varGradient_equivalenvce [source]

```
lemma totalTimeDerivative_varGradient_equivalenvce [CompleteSpace X] (L L' : Time → X → X → ℝ)
    (htot : IsTotalTimeDerivative (L' - L))
    (q : Time → X) (hq : ContDiff ℝ ∞ q):
    (δ (q':=q), ∫ t, L' t (q' t) (fderiv ℝ q' t 1)) =
       (δ (q':=q), ∫ t, L t (q' t) (fderiv ℝ q' t 1))
```

lemma totalTimeDerivative_eulerLagrange_equivalenvce [source]

Corollary: If L and L' differ by a total time derivative, then the corresponding Euler-Lagrange operators coincide

```
lemma totalTimeDerivative_eulerLagrange_equivalenvce [CompleteSpace X] (L L' : Time → X → X →
    ℝ)
    (htot : IsTotalTimeDerivative (L' - L)) (hContDiff : (ContDiff ℝ ∞ ₁L) ∨ (ContDiff ℝ ∞ ₁L')
    )
    (q : Time → X) (hq : ContDiff ℝ ∞ q) : eulerLagrangeOp L q = eulerLagrangeOp L' q
```

2.7 Physlib.ClassicalMechanics.RigidBody.AngularMomentum

Physlib/ClassicalMechanics/RigidBody/AngularMomentum.lean on GitHub.

def angularMomentum [source]

The angular momentum `L = ∫ r × (ω × r) dm` of a rigid body rotating with angular velocity `ω` about its reference point (each body point at `r` moving with velocity `ω × r`). Its `i`-th component is `ρ` applied to the scalar field `[r × (ω × r)]ᵢ`.

```
noncomputable def angularMomentum (R : RigidBody 3) (ω : Fin 3 → ℝ) : Fin 3 → ℝ
```

theorem angularMomentum_eq_inertiaTensor_mulVec [source]

The angular momentum of a rigid body equals its inertia tensor applied to the angular velocity: `L = I ω`.

```
theorem angularMomentum_eq_inertiaTensor_mulVec (R : RigidBody 3) (ω : Fin 3 → ℝ) :
    R.angularMomentum ω = R.inertiaTensor *ᵥ ω
```

2.8 Physlib.ClassicalMechanics.RigidBody.AngularVelocity

Physlib/ClassicalMechanics/RigidBody/AngularVelocity.lean on GitHub.

def angularVelocityTensor [source]

The angular velocity tensor `Ω(t) = Ṙ(t) R(t)ᵀ` of a rigid body in motion, where `R(t) = orientation t`. It is the antisymmetric tensor `Ω` in the Landau–Lifshitz decomposition `v = V + Ω × r` of the velocity of a point of the body.

```
noncomputable def angularVelocityTensor (M : RigidBodyMotion d) (t : Time) :
    Matrix (Fin d) (Fin d) ℝ
```

lemma angularVelocityTensor_eq [source]

```
lemma angularVelocityTensor_eq (M : RigidBodyMotion d) (t : Time) :
    M.angularVelocityTensor t = ∂ₜ (fun s => (M.orientation s).1) t * ((M.orientation t).1)ᵀ
```

lemma angularVelocityTensor_transpose [source]

The angular velocity tensor is skew-symmetric, `Ωᵀ = -Ω`: it lies in the Lie algebra `𝔰𝔬(d)`. This is the litmus check that `Ω = Ṙ Rᵀ` is a genuine angular-velocity tensor, and follows by differentiating the orthogonality identity `R Rᵀ = 1`.

```
lemma angularVelocityTensor_transpose (M : RigidBodyMotion d) (t : Time)
    (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t) :
    (M.angularVelocityTensor t)ᵀ = - M.angularVelocityTensor t
```

lemma angularVelocityTensor_of_orientation_const [source]

A rigid body whose orientation is constant in time has zero angular velocity.

```
lemma angularVelocityTensor_of_orientation_const (M : RigidBodyMotion d)
    (R : Matrix.specialOrthogonalGroup (Fin d) ℝ) (h : M.orientation = fun _ => R) :
    M.angularVelocityTensor = 0
```

def angularVelocity [source]

*The angular velocity *vector* `ω(t)` of a rigid body moving in three-dimensional space: the vector dual to the angular velocity tensor `Ω(t)` under the hat map, `ω = Ωᵛ`. It is the angular velocity `ω` in the Landau–Lifshitz decomposition `v = V + ω × r` of the velocity of a point of the body, where `ω × r` is the cross product with `ω`.*

```
noncomputable def angularVelocity (M : RigidBodyMotion 3) (t : Time) : Fin 3 → ℝ
```

lemma angularVelocity_eq [source]

```
lemma angularVelocity_eq (M : RigidBodyMotion 3) (t : Time) :
    M.angularVelocity t = crossProductVee (M.angularVelocityTensor t)
```

lemma crossProductMatrix_angularVelocity [source]

The hat map recovers the angular velocity tensor from the angular velocity vector, `[ω]ₓ = Ω`. This is the defining relationship between the vector and tensor forms of the angular velocity in three dimensions; it holds because `Ω` is skew-symmetric.

```
lemma crossProductMatrix_angularVelocity (M : RigidBodyMotion 3) (t : Time)
    (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t) :
```

```
    crossProductMatrix (M.angularVelocity t) = M.angularVelocityTensor t
```

lemma angularVelocity_of_orientation_const [source]

A rigid body whose orientation is constant in time has zero angular velocity vector.

```
lemma angularVelocity_of_orientation_const (M : RigidBodyMotion 3)
    (R : Matrix.specialOrthogonalGroup (Fin 3) ℝ) (h : M.orientation = fun _ => R) :
    M.angularVelocity = 0
```

lemma angularVelocityTensor_mul_orientation [source]

The time derivative of the orientation is $\dot{R} = \Omega R$, recovering the orientation path from its angular velocity tensor $\Omega = \dot{R} R^T$ via the orthogonality $R^T R = 1$.

```
lemma angularVelocityTensor_mul_orientation (M : RigidBodyMotion d) (t : Time) :
    M.angularVelocityTensor t * (M.orientation t).1 = ∂ₜ (fun s => (M.orientation s).1) t
```

lemma deriv_orientation_mulVec_eq_angularVelocity_cross [source]

The rotational part of the velocity of the body point `y` is the cross product of the angular velocity with the point's position relative to the centre of mass: `Ṙ(t) (y − c) = ω(t) × (displacement t y − comTrajectory t)`.

```
lemma deriv_orientation_mulVec_eq_angularVelocity_cross (M : RigidBodyMotion 3) (y : Space 3)
    (t : Time) (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t) :
    ∂ₜ (fun s => (M.orientation s).1) t *ᵥ (fun j => y j - M.centerOfMass j)
      = M.angularVelocity t ×₃ fun j => M.displacement t y j - M.comTrajectory t j
```

theorem velocity_eq_angularVelocity [source]

The Landau–Lifshitz velocity decomposition `v = V + ω × r` for a rigid body in three dimensions: the velocity of a body point is the centre-of-mass velocity plus the cross product of the angular velocity with the point's position relative to the centre of mass.

```
theorem velocity_eq_angularVelocity (M : RigidBodyMotion 3) (y : Space 3) (t : Time) (i : Fin
    3)
    (hR : Differentiable ℝ (fun s => (M.orientation s).1)) (hX : Differentiable ℝ M.
    comTrajectory) :
    M.velocity y t i = M.centerOfMassVelocity t i
        + (M.angularVelocity t ×₃ fun j => M.displacement t y j - M.comTrajectory t j) i
```

def bodyAngularVelocityTensor [source]

The body-frame (co-rotating) angular velocity tensor $\Omega_{body}(t) = R(t)^T \dot{R}(t)$ of a rigid body in motion, where `R(t) = orientation t`.

```
noncomputable def bodyAngularVelocityTensor (M : RigidBodyMotion d) (t : Time) :
    Matrix (Fin d) (Fin d) ℝ
```

lemma bodyAngularVelocityTensor_eq [source]

```
lemma bodyAngularVelocityTensor_eq (M : RigidBodyMotion d) (t : Time) :
```

```
    M.bodyAngularVelocityTensor t = ((M.orientation t).1)ᵀ * ∂ₜ (fun s => (M.orientation s).1)
    t
```

lemma orientation_mul_bodyAngularVelocityTensor [source]

The time derivative of the orientation is $\dot{R}$ = R Ω_body, recovering the orientation path from its body-frame angular velocity tensor via the orthogonality R R^T = 1.

```
lemma orientation_mul_bodyAngularVelocityTensor (M : RigidBodyMotion d) (t : Time) :
    (M.orientation t).1 * M.bodyAngularVelocityTensor t = ∂ₜ (fun s => (M.orientation s).1) t
```

lemma angularVelocityTensor_eq_orientation_conj [source]

The spatial tensor is the body-frame tensor rotated into the inertial frame, Ω = R Ω_body R^T.

```
lemma angularVelocityTensor_eq_orientation_conj (M : RigidBodyMotion d) (t : Time) :
    M.angularVelocityTensor t
      = (M.orientation t).1 * M.bodyAngularVelocityTensor t * ((M.orientation t).1)ᵀ
```

lemma bodyAngularVelocityTensor_eq_orientation_conj [source]

The body-frame tensor is the spatial tensor conjugated into the body frame, Ω_body = R^T Ω R.

```
lemma bodyAngularVelocityTensor_eq_orientation_conj (M : RigidBodyMotion d) (t : Time) :
    M.bodyAngularVelocityTensor t
      = ((M.orientation t).1)ᵀ * M.angularVelocityTensor t * (M.orientation t).1
```

lemma bodyAngularVelocityTensor_transpose [source]

The body-frame angular velocity tensor is skew-symmetric, Ω_body^T = -Ω_body: it lies in the Lie algebra 𝔰𝔬(d), inherited from the spatial tensor by conjugation.

```
lemma bodyAngularVelocityTensor_transpose (M : RigidBodyMotion d) (t : Time)
    (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t) :
    (M.bodyAngularVelocityTensor t)ᵀ = - M.bodyAngularVelocityTensor t
```

def bodyAngularVelocity [source]

*The body-frame angular velocity *vector* ω_body(t) of a rigid body moving in three-dimensional space: the vector dual to the body-frame angular velocity tensor Ω_body(t) under the hat map, ω_body = Ω_body^v.*

```
noncomputable def bodyAngularVelocity (M : RigidBodyMotion 3) (t : Time) : Fin 3 → ℝ
```

lemma bodyAngularVelocity_eq [source]

```
lemma bodyAngularVelocity_eq (M : RigidBodyMotion 3) (t : Time) :
    M.bodyAngularVelocity t = crossProductVee (M.bodyAngularVelocityTensor t)
```

lemma crossProductMatrix_bodyAngularVelocity [source]

The hat map recovers the body-frame angular velocity tensor from the body-frame angular velocity vector, `[ω_body]ₓ = Ω_body`; it holds because `Ω_body` is skew-symmetric.

```
lemma crossProductMatrix_bodyAngularVelocity (M : RigidBodyMotion 3) (t : Time)
    (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t) :
    crossProductMatrix (M.bodyAngularVelocity t) = M.bodyAngularVelocityTensor t
```

lemma bodyAngularVelocityTensor_of_orientation_const [source]

A rigid body whose orientation is constant in time has zero body-frame angular velocity tensor.

```
lemma bodyAngularVelocityTensor_of_orientation_const (M : RigidBodyMotion d)
    (R : Matrix.specialOrthogonalGroup (Fin d) ℝ) (h : M.orientation = fun _ => R) :
    M.bodyAngularVelocityTensor = 0
```

lemma bodyAngularVelocity_of_orientation_const [source]

A rigid body whose orientation is constant in time has zero body-frame angular velocity vector.

```
lemma bodyAngularVelocity_of_orientation_const (M : RigidBodyMotion 3)
    (R : Matrix.specialOrthogonalGroup (Fin 3) ℝ) (h : M.orientation = fun _ => R) :
    M.bodyAngularVelocity = 0
```

lemma deriv_orientation_mulVec_eq_orientation_bodyAngularVelocity_cross [source]

The rotational velocity written through the body-frame angular velocity, `Ṙ(t) v = R(t) (ω_body(t) × v)`: the derivative of the orientation acts as the body-frame angular velocity `ω_body` crossed with `v`, rotated back into the inertial frame by `R`.

```
lemma deriv_orientation_mulVec_eq_orientation_bodyAngularVelocity_cross (M : RigidBodyMotion 3)
    (v : Fin 3 → ℝ) (t : Time)
    (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t) :
    ∂ₜ (fun s => (M.orientation s).1) t *ᵥ v
      = (M.orientation t).1 *ᵥ (M.bodyAngularVelocity t ×₃ v)
```

2.9 Physlib.ClassicalMechanics.RigidBody.Basic

Physlib/ClassicalMechanics/RigidBody/Basic.lean on GitHub.

lemma rho_one [source]

The mass distribution applied to the constant function `1` is the total mass.

```
@[simp]
lemma rho_one {d : ℕ} (R : RigidBody d) :
    R.ρ (1 : C^∞⟮ℐ(ℝ, Space d), Space d; ℐ(ℝ, ℝ), ℝ⟯) = R.mass
```

def cmap [source]

Bundle a smooth real-valued function on `Space d` as an element of the space of test functions. Keeping this as a named constructor ensures the resulting type head stays `ContMDiffMap`, so the module/ring operations and `comp` resolve correctly.

```
def cmap {d : ℕ} (f : Space d → ℝ) (hf : ContDiff ℝ ⊤ f) :
    C^⊤ ⟮𝓘(ℝ, Space d), Space d; 𝓘(ℝ, ℝ), ℝ⟯
```

lemma cmap_apply [source]

```
@[simp]
lemma cmap_apply {d : ℕ} (f : Space d → ℝ) (hf : ContDiff ℝ ⊤ f) (y : Space d) :
    cmap f hf y = f y
```

lemma rho_coord_sub_centerOfMass [source]

The first moment of the mass distribution about its own centre of mass vanishes: for nonzero mass, `ρ` of the centred `j`-th coordinate function is zero.

```
lemma rho_coord_sub_centerOfMass {d : ℕ} (R : RigidBody d) (h : R.mass ≠ 0) (j : Fin d) :
    R.ρ (cmap (fun y => y j - R.centerOfMass j) (by fun_prop)) = 0
```

2.10 Physlib.ClassicalMechanics.RigidBody.KineticEnergy

Physlib/ClassicalMechanics/RigidBody/KineticEnergy.lean on GitHub.

def rotationalKineticEnergy [source]

The rotational kinetic energy of a rigid body rotating with angular velocity `ω` about its reference point: half the contraction of `ω` with the inertia tensor, `T = ½ ω · (I ω)`.

```
noncomputable def rotationalKineticEnergy (R : RigidBody 3) (ω : Fin 3 → ℝ) : ℝ
```

lemma rotationalKineticEnergy_eq_angularMomentum [source]

The rotational kinetic energy is half the contraction of the angular velocity with the angular momentum: `T = ½ ω · L`.

```
lemma rotationalKineticEnergy_eq_angularMomentum (R : RigidBody 3) (ω : Fin 3 → ℝ) :
    R.rotationalKineticEnergy ω = (1 / (2 : ℝ)) * (ω ⬝ᵥ R.angularMomentum ω)
```

theorem rotationalKineticEnergy_eq_integral [source]

The rotational kinetic energy equals the mass integral of the local rotational speed squared: `T = ½ ∫ |ω × r|² dm`.

```
theorem rotationalKineticEnergy_eq_integral (R : RigidBody 3) (ω : Fin 3 → ℝ) :
    R.rotationalKineticEnergy ω
      = (1 / (2 : ℝ)) * R.ρ ⟨fun x => (ω ×₃ (x : Fin 3 → ℝ)) ⬝ᵥ (ω ×₃ (x : Fin 3 → ℝ)),
        ContDiff.contMDiff <| (contDiff_cross_dotProduct_cross ω).comp
          (contDiff_pi.mpr fun i => Space.eval_contDiff i)⟩
```

def kineticEnergy [source]

The total kinetic energy of a rigid body in motion at time `t`: half the mass integral of the squared speed of the body points, `T = ½ ∫ ⟪v, v⟫ dm`, with the point velocity taken in the closed form `velocityClosedForm`.

```
noncomputable def kineticEnergy {d : ℕ} (M : RigidBodyMotion d) (t : Time) : ℝ
```

lemma kineticEnergy_eq_integral_velocity [source]

For a differentiable motion the integrand of `kineticEnergy` is the squared speed of the honest point velocity: `T = ½ ∫ ⟪v, v⟫ dm` with `v = ∂ₜ (displacement · y)`.

```
lemma kineticEnergy_eq_integral_velocity {d : ℕ} (M : RigidBodyMotion d) (t : Time)
    (hR : Differentiable ℝ (fun s => (M.orientation s).1))
    (hX : Differentiable ℝ M.comTrajectory) :
    M.kineticEnergy t = (1 / (2 : ℝ)) * M.ρ (cmap
      (fun y => (⟪M.velocity y t, M.velocity y t⟫_ℝ))
      (by
        simp only [← M.velocityClosedForm_eq_velocity t hR hX]
        exact M.contDiff_velocityClosedForm_inner t))
```

lemma kineticEnergy_integrand_split [source]

The squared-speed integrand of `kineticEnergy` splits into the squared rotational speed `|Ṙ (y − c)|²`, a term linear in the body-frame coordinate `y − c`, and the constant squared centre-of-mass speed `⟪V, V⟫`.

```
lemma kineticEnergy_integrand_split {d : ℕ} (M : RigidBodyMotion d) (t : Time) :
    cmap (fun y => (⟪M.velocityClosedForm t y, M.velocityClosedForm t y⟫_ℝ))
        (M.contDiff_velocityClosedForm_inner t)
      = cmap (fun y =>
            (∂ₜ (fun s => (M.orientation s).1) t *ᵥ fun j => y j - M.centerOfMass j) ⋅ᵥ
            (∂ₜ (fun s => (M.orientation s).1) t *ᵥ fun j => y j - M.centerOfMass j))
          (by simp only [dotProduct, Matrix.mulVec]; fun_prop)
        + ∑ j, (2 * (((M.centerOfMassVelocity t : Fin d → ℝ) ᵥ*
              ∂ₜ (fun s => (M.orientation s).1) t) j)) •
            cmap (fun y => y j - M.centerOfMass j) (by fun_prop)
        + (⟪M.centerOfMassVelocity t, M.centerOfMassVelocity t⟫_ℝ) •
            (1 : C^T⌊ℐ(ℝ, Space d), Space d; ℐ(ℝ, ℝ), ℝ⌋)
```

theorem kineticEnergy_eq_translational_add_rotational [source]

***König's theorem**, general form: the total kinetic energy of a rigid body in motion splits into the kinetic energy of the centre of mass plus the rotational energy about the centre of mass, `T = ½ M ⟪V, V⟫ + ½ ∫ |Ṙ (y − c)|² dm`.*

```
theorem kineticEnergy_eq_translational_add_rotational {d : ℕ} (M : RigidBodyMotion d) (t : Time
    )
    (h : M.mass ≠ 0) :
    M.kineticEnergy t
      = (1 / (2 : ℝ)) * M.mass * (⟪M.centerOfMassVelocity t, M.centerOfMassVelocity t⟫_ℝ)
        + (1 / (2 : ℝ)) * M.ρ (cmap (fun y =>
            (∂ₜ (fun s => (M.orientation s).1) t *ᵥ fun j => y j - M.centerOfMass j) ⋅ᵥ
            (∂ₜ (fun s => (M.orientation s).1) t *ᵥ fun j => y j - M.centerOfMass j))
          (by simp only [dotProduct, Matrix.mulVec]; fun_prop))
```

theorem kineticEnergy_eq_translational_add_angularVelocity [source]

***König's theorem** in three dimensions: the total kinetic energy of a rigid body in motion splits as `T = ½ M ⟪V, V⟫ + ½ ∫ |ω × r|² dm`, with `ω` the angular velocity vector and `r = displacement − comTrajectory` the position of the body point relative to the centre of mass.*

```
theorem kineticEnergy_eq_translational_add_angularVelocity (M : RigidBodyMotion 3) (t : Time)
    (h : M.mass ≠ 0) (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t) :
    M.kineticEnergy t
      = (1 / (2 : ℝ)) * M.mass * (⟪M.centerOfMassVelocity t, M.centerOfMassVelocity t⟫_ℝ)
        + (1 / (2 : ℝ)) * M.ρ (cmap (fun y =>
            (M.angularVelocity t ×₃ fun j => M.displacement t y j - M.comTrajectory t j) ·ᵥ
            (M.angularVelocity t ×₃ fun j => M.displacement t y j - M.comTrajectory t j))
          (by
            exact (contDiff_cross_dotProduct_cross (M.angularVelocity t)).comp
              (contDiff_pi.mpr fun j =>
                ((Space.eval_contDiff j).comp (M.displacement_contDiff t)).sub
                  contDiff_const)))
```

theorem kineticEnergy_eq_translational_add_bodyAngularVelocity [source]

***König's theorem** in the body frame. The total kinetic energy `M.kineticEnergy t`, formed from the lab-frame point velocities, splits at the centre of mass (`centerOfMass = 0`) as `T = ½ M ⟪V, V⟫ + rotationalKineticEnergy ω_body`. The rotational energy is a frame-independent scalar, so it is evaluated here from the *body-frame* angular velocity `ω_body`: the spatial form `|Ṙ (y − c)|²` and the body form `|ω_body × (y − c)|²` agree because `Ṙ (y − c) = R (ω_body × (y − c))` and `R`, being orthogonal, preserves the norm.*

```
theorem kineticEnergy_eq_translational_add_bodyAngularVelocity (M : RigidBodyMotion 3) (t :
    Time)
    (h : M.mass ≠ 0) (hR : DifferentiableAt ℝ (fun s => (M.orientation s).1) t)
    (hc : M.centerOfMass = 0) :
    M.kineticEnergy t
      = (1 / (2 : ℝ)) * M.mass * (⟪M.centerOfMassVelocity t, M.centerOfMassVelocity t⟫_ℝ)
        + M.toRigidBody.rotationalKineticEnergy (M.bodyAngularVelocity t)
```

2.11 Physlib.ClassicalMechanics.RigidBody.Motion

Physlib/ClassicalMechanics/RigidBody/Motion.lean on GitHub.

structure RigidBodyMotion [source]

A motion of a rigid body in `d`-dimensional space: the body together with the inertial-frame trajectory of its centre of mass and its time-dependent orientation (a rotation about the centre of mass).

```
structure RigidBodyMotion (d : ℕ) extends RigidBody d where
  comTrajectory : Time → Space d
  orientation : Time → Matrix.specialOrthogonalGroup (Fin d) ℝ
```

lemma orientation_mul_transpose [source]

The orientation matrix is special orthogonal, so `R Rᵀ = 1`.

```
lemma orientation_mul_transpose {d : ℕ} (M : RigidBodyMotion d) (t : Time) :
    (M.orientation t).1 * ((M.orientation t).1)^T = 1
```

def centerOfMassVelocity [source]

The velocity of the centre of mass of a rigid body in motion, defined as the time-derivative of its centre-of-mass trajectory. This is the velocity `V` in the Landau–Lifshitz decomposition `v = V + Ω × r` of the velocity of a point of the body.

```
noncomputable def centerOfMassVelocity {d : ℕ} (M : RigidBodyMotion d) : Time → Space d
```

lemma centerOfMassVelocity_eq [source]

```
lemma centerOfMassVelocity_eq {d : ℕ} (M : RigidBodyMotion d) :
    M.centerOfMassVelocity = ∂ₜ M.comTrajectory
```

lemma centerOfMassVelocity_of_comTrajectory_const [source]

A rigid body whose centre of mass is stationary has zero centre-of-mass velocity.

```
lemma centerOfMassVelocity_of_comTrajectory_const {d : ℕ} (M : RigidBodyMotion d) (c : Space d)
    (h : M.comTrajectory = fun _ => c) : M.centerOfMassVelocity = 0
```

def linearMomentum [source]

The linear momentum of a rigid body in motion: the total mass times the velocity of the centre of mass.

```
noncomputable def linearMomentum {d : ℕ} (M : RigidBodyMotion d) : Time → Space d
```

lemma linearMomentum_eq [source]

```
lemma linearMomentum_eq {d : ℕ} (M : RigidBodyMotion d) :
    M.linearMomentum = fun t => M.mass • M.centerOfMassVelocity t
```

def displacement [source]

The rigid displacement carrying the body frame into the inertial frame at time `t`: the rotation `orientation t` about the centre of mass, followed by the translation placing the centre of mass at `comTrajectory t`.

```
noncomputable def displacement {d : ℕ} (M : RigidBodyMotion d) (t : Time) : Space d → Space d
```

lemma displacement_apply [source]

The `k`-th coordinate of the rigid displacement applied to `y`.

```
lemma displacement_apply {d : ℕ} (M : RigidBodyMotion d) (t : Time) (y : Space d) (k : Fin d) :
    M.displacement t y k =
      (∑ j, (M.orientation t).val k j * (y j - M.centerOfMass j)) + M.comTrajectory t k
```

lemma displacement_contDiff [source]

```
lemma displacement_contDiff {d : ℕ} (M : RigidBodyMotion d) (t : Time) :
    ContDiff ℝ ⊤ (M.displacement t)
```

lemma orientation_mulVec_sub_centerOfMass [source]

Rotating the body-frame position of `y` relative to the centre of mass gives its inertial-frame position relative to the moving centre of mass: `R(t) (y − c) = displacement t y − comTrajectory t`.

```
lemma orientation_mulVec_sub_centerOfMass {d : ℕ} (M : RigidBodyMotion d) (t : Time)
    (y : Space d) :
    (M.orientation t).1 *ᵥ (fun j => y j - M.centerOfMass j)
      = fun j => M.displacement t y j - M.comTrajectory t j
```

def massDistribution [source]

The mass distribution of the rigid body in motion at time `t`: the pushforward of the body-fixed mass distribution along the rigid displacement, acting on a test function `f` by `f ↦ ρ (f ∘ displacement t)`.

```
noncomputable def massDistribution {d : ℕ} (M : RigidBodyMotion d) (t : Time) : RigidBody d
    where
```

lemma massDistribution_mass [source]

The motion preserves the total mass: the mass distribution at any time `t` has the same total mass as the body.

```
@[simp]
lemma massDistribution_mass {d : ℕ} (M : RigidBodyMotion d) (t : Time) :
    (M.massDistribution t).mass = M.mass
```

lemma contMDiffMap_sum_apply [source]

Evaluation commutes with finite sums of smooth functions.

```
private lemma contMDiffMap_sum_apply {d : ℕ} {ι : Type*} (s : Finset ι)
    (f : ι → C^⊤⟮𝓘(ℝ, Space d), Space d; 𝓘(ℝ, ℝ), ℝ⟯) (y : Space d) :
    (∑ j ∈ s, f j) y = ∑ j ∈ s, f j y
```

lemma massDistribution_centerOfMass [source]

The centre of mass of the moving mass distribution tracks the prescribed trajectory: for a body of nonzero mass, the centre of mass of `massDistribution M t` is exactly `comTrajectory t`. This is the decisive check that `comTrajectory` and `orientation` are wired correctly in `RigidBodyMotion`.

```
lemma massDistribution_centerOfMass {d : ℕ} (M : RigidBodyMotion d) (t : Time) (h : M.mass ≠ 0)
    :
    (M.massDistribution t).centerOfMass = M.comTrajectory t
```

def velocity [source]

The velocity of the material point `y` of a rigid body in motion: the inertial-frame time derivative of the trajectory `s ↦ displacement s y` of that point.

```
noncomputable def velocity {d : ℕ} (M : RigidBodyMotion d) (y : Space d) : Time → Space d
```

lemma velocity_eq [source]

```
lemma velocity_eq {d : ℕ} (M : RigidBodyMotion d) (y : Space d) (t : Time) :
    M.velocity y t = ∂ₜ (fun s => M.displacement s y) t
```

lemma velocity_apply [source]

The `i`-th component of the velocity of a body point is the time derivative of the `i`-th coordinate of its inertial-frame trajectory.

```
lemma velocity_apply {d : ℕ} (M : RigidBodyMotion d) (y : Space d) (t : Time) (i : Fin d)
    (hd : Differentiable ℝ (fun s => M.displacement s y)) :
    M.velocity y t i = ∂ₜ (fun s => M.displacement s y i) t
```

lemma velocity_centerOfMass [source]

The material point at the centre of mass moves with the centre-of-mass velocity, for any motion: `v(centreOfMass) = V`. This is the velocity counterpart of `massDistribution_centerOfMass`.

```
lemma velocity_centerOfMass {d : ℕ} (M : RigidBodyMotion d) :
    M.velocity M.centerOfMass = M.centerOfMassVelocity
```

lemma velocity_of_orientation_const [source]

A rigid body in pure translation (constant orientation) has every point moving with the centre-of-mass velocity: `v = V`.

```
lemma velocity_of_orientation_const {d : ℕ} (M : RigidBodyMotion d) (y : Space d)
    (R : Matrix.specialOrthogonalGroup (Fin d) ℝ) (h : M.orientation = fun _ => R) :
    M.velocity y = M.centerOfMassVelocity
```

lemma velocity_eq_deriv_orientation [source]

The velocity of a body point decomposes as `v = Ṙ (y − c) + V`: the rate of change of the orientation acting on the body-frame position, plus the centre-of-mass velocity.

```
lemma velocity_eq_deriv_orientation {d : ℕ} (M : RigidBodyMotion d) (y : Space d) (t : Time)
    (i : Fin d) (hR : Differentiable ℝ (fun s => (M.orientation s).1))
    (hX : Differentiable ℝ M.comTrajectory) :
    M.velocity y t i
      = (∂ₜ (fun s => (M.orientation s).1) t *ᵥ fun j => y j - M.centerOfMass j) i
        + M.centerOfMassVelocity t i
```

def velocityClosedForm [source]

The closed form `Ṙ(t) (y − c) + V(t)` of the velocity of the body point `y` at time `t`. Unlike `velocity` — whose junk values on non-differentiable motions need not vary

continuously with `y` — it is polynomial in `y` for any motion, so its squared speed can be bundled as the smooth integrand of the total kinetic energy; for differentiable motions the two agree, see `velocityClosedForm_eq_velocity`.

```
noncomputable def velocityClosedForm {d : ℕ} (M : RigidBodyMotion d) (t : Time) (y : Space d) :
    Space d
```

lemma `velocityClosedForm_apply` [source]

The `i`-th coordinate of the closed-form velocity.

```
lemma velocityClosedForm_apply {d : ℕ} (M : RigidBodyMotion d) (t : Time) (y : Space d)
    (i : Fin d) :
    M.velocityClosedForm t y i
      = (∂ₜ (fun s => (M.orientation s).1) t *ᵥ fun j => y j - M.centerOfMass j) i
        + M.centerOfMassVelocity t i
```

lemma `velocityClosedForm_val` [source]

The closed-form velocity as a plain vector-valued function of the coordinates of `y`.

```
lemma velocityClosedForm_val {d : ℕ} (M : RigidBodyMotion d) (t : Time) (y : Space d) :
    (M.velocityClosedForm t y : Fin d → ℝ)
      = (∂ₜ (fun s => (M.orientation s).1) t *ᵥ fun j => y j - M.centerOfMass j)
        + (M.centerOfMassVelocity t : Fin d → ℝ)
```

lemma `velocityClosedForm_eq_velocity` [source]

For a differentiable motion the closed-form velocity is the honest point velocity.

```
lemma velocityClosedForm_eq_velocity {d : ℕ} (M : RigidBodyMotion d) (t : Time)
    (hR : Differentiable ℝ (fun s => (M.orientation s).1))
    (hX : Differentiable ℝ M.comTrajectory) (y : Space d) :
    M.velocityClosedForm t y = M.velocity y t
```

lemma `contDiff_velocityClosedForm_inner` [source]

The squared speed of a body point, in closed form, is a smooth function of the point.

```
lemma contDiff_velocityClosedForm_inner {d : ℕ} (M : RigidBodyMotion d) (t : Time) :
    ContDiff ℝ ⊤ fun y : Space d => (⟪M.velocityClosedForm t y, M.velocityClosedForm t y⟫_ℝ)
```

2.12 `Physlib.CondensedMatter.Thermoelectric.Basic`

`Physlib/CondensedMatter/Thermoelectric/Basic.lean` on GitHub.

structure `ThermoelectricMaterial` [source]

A thermoelectric material in the linear-response regime, characterized by its four transport coefficients.

```
structure ThermoelectricMaterial where
  S : ℝ
  σ : ℝ
  κl : ℝ
  κe : ℝ
```

```
  σ_pos : 0 < σ
  κl_pos : 0 < κl
  κe_nonneg : 0 ≤ κe
```

def powerFactor [source]

The thermoelectric power factor `PF = σ S²` of a material.

```
def powerFactor (M : ThermoelectricMaterial) : ℝ
```

lemma powerFactor_pos [source]

The power factor is positive when the Seebeck coefficient is nonzero.

```
lemma powerFactor_pos {M : ThermoelectricMaterial} (hS : M.S ≠ 0) :
    0 < M.powerFactor
```

def totalThermalConductivity [source]

The total thermal conductivity `κl + κe` of a material, the sum of the lattice (phonon) and electronic contributions.

```
def totalThermalConductivity (M : ThermoelectricMaterial) : ℝ
```

lemma totalThermalConductivity_pos [source]

The total thermal conductivity is positive.

```
lemma totalThermalConductivity_pos (M : ThermoelectricMaterial) :
    0 < M.totalThermalConductivity
```

def figureOfMerit [source]

The dimensionless thermoelectric figure of merit `zT = PF · T / (κl + κe)` of a material at absolute temperature `T`.

```
def figureOfMerit (M : ThermoelectricMaterial) (T : ℝ) : ℝ
```

lemma figureOfMerit_eq [source]

The figure of merit in its standard flat form `zT = σ S² T / (κl + κe)`.

```
lemma figureOfMerit_eq (M : ThermoelectricMaterial) (T : ℝ) :
    M.figureOfMerit T = M.σ * M.S ^ 2 * T / (M.κl + M.κe)
```

lemma figureOfMerit_pos [source]

The figure of merit is positive at positive temperature when the Seebeck coefficient is nonzero.

```
lemma figureOfMerit_pos {M : ThermoelectricMaterial} {T : ℝ}
    (hS : M.S ≠ 0) (hT : 0 < T) :
    0 < M.figureOfMerit T
```

lemma figureOfMerit_le_of_le [source]

Lowering the lattice thermal conductivity raises the figure of merit: if `M.κl ≤ κl'` then the material with lattice conductivity `κl'` (all other coefficients equal) has the smaller `zT`. This is the phonon-glass electron-crystal design principle: scatter phonons without degrading electronic transport. Note the hypotheses: no condition on the Seebeck coefficient is needed, since monotonicity only requires the numerator `σ S² T` to be nonnegative.

```
lemma figureOfMerit_le_of_le (M : ThermoelectricMaterial) {κl' T : ℝ}
    (hpos : 0 < κl') (h : M.κl ≤ κl') (hT : 0 ≤ T) :
    figureOfMerit { M with κl := κl', κl_pos := hpos } T ≤ M.figureOfMerit T
```

2.13 Physlib.FluidDynamics.CauchyFlow.Basic

Physlib/FluidDynamics/CauchyFlow/Basic.lean on GitHub.

structure CauchyFlow [source]

A fluid flow equipped with Cauchy stress and specific body-force fields.

These are the additional independent fields needed to state a local momentum balance. The Cauchy stress represents contact forces, and the specific body force represents volumetric force per unit mass. Pressure and viscosity enter through stress laws rather than as redundant fields of `CauchyFlow` itself.

```
structure CauchyFlow (d : ℕ) extends FluidFlow d where
  stress : StressTensor d
  specificBodyForce : VectorField d
```

2.14 Physlib.FluidDynamics.CauchyFlow.BodyForce

Physlib/FluidDynamics/CauchyFlow/BodyForce.lean on GitHub.

def HasBodyForcePotential [source]

A flow has body-force potential `Phi` when its specific body force is minus the gradient of `Phi`.

```
def HasBodyForcePotential (d : ℕ) (flow : CauchyFlow d) (potential : Space d → ℝ) : Prop
```

def HasConservativeBodyForce [source]

A flow has conservative body force when its specific body force has some potential.

```
def HasConservativeBodyForce (d : ℕ) (flow : CauchyFlow d) : Prop
```

2.15 Physlib.FluidDynamics.CauchyFlow.Inviscid

Physlib/FluidDynamics/CauchyFlow/Inviscid.lean on GitHub.

def IsInviscid [source]

A Cauchy flow is inviscid with pressure `p` when its stress is the isotropic pressure stress `-p I`.

In this formulation, inviscid means that the Cauchy stress has no viscous or shear contribution. It does not rule out body forces; those are carried separately by `CauchyFlow.specificBodyForce`.

```
def IsInviscid (d : ℕ) (flow : CauchyFlow d) (pressure : ScalarField d) : Prop
```

lemma matrixDiv_inviscid_pressure_stress [source]

The matrix divergence of the inviscid pressure stress `-p I` is `-grad p`.

```
lemma matrixDiv_inviscid_pressure_stress (d : ℕ) (pressureAtTime : Space d → ℝ) :
    matrixDiv d (fun x => (-(pressureAtTime x)) • (1 : Matrix (Fin d) (Fin d) ℝ)) =
      -∇ pressureAtTime
```

theorem matrixDiv_stress_eq_neg_grad_pressure_of_is_inviscid [source]

In an inviscid Cauchy flow, the stress-divergence force is the usual negative pressure-gradient term.

```
theorem matrixDiv_stress_eq_neg_grad_pressure_of_is_inviscid
    (d : ℕ) (flow : CauchyFlow d) (pressure : ScalarField d)
    (hInviscid : IsInviscid d flow pressure) :
    ∀ t, matrixDiv d (flow.stress t) = -∇ (pressure t)
```

2.16 Physlib.FluidDynamics.CauchyFlow.Momentum

Physlib/FluidDynamics/CauchyFlow/Momentum.lean on GitHub.

def CauchyMomentumEquation [source]

Conservation of momentum in conservative matrix-divergence form.

The equation is

`partial_t (rho u) + matrixDiv (rho u ⊗ u) = matrixDiv sigma + rho f`.

Here `stress` is intentionally not yet specialized to a constitutive law.

```
def CauchyMomentumEquation (d : ℕ) (flow : CauchyFlow d) : Prop
```

def ConvectiveCauchyMomentumEquation [source]

Conservation of momentum in convective form.

The equation is

`rho (partial_t u + (u · ∇)u) = matrixDiv sigma + rho f`.

Here `stress` is intentionally not yet specialized to a constitutive law.

```
def ConvectiveCauchyMomentumEquation (d : ℕ) (flow : CauchyFlow d) : Prop
```

lemma cauchyMomentumEquation_iff_convectiveCauchyMomentumEquation [source]

The conservative and convective Cauchy momentum equations are equivalent when the classical continuity equation holds.

The differentiability assumptions are exactly the product-rule assumptions used to rewrite `partial_t (rho u)` and `matrixDiv (rho u ⊗ u)`.

```
lemma cauchyMomentumEquation_iff_convectiveCauchyMomentumEquation
    (d : ℕ) (flow : CauchyFlow d)
    (hContinuity : FluidFlow.ClassicalContinuityEquation d flow.toFluidFlow)
    (hRhoTime : ∀ t x, DifferentiableAt ℝ (flow.rho · x) t)
    (hVelocityTime : ∀ t x, DifferentiableAt ℝ (flow.velocity · x) t)
    (hMomentumDensity : ∀ t, Differentiable ℝ (FluidFlow.momentumDensity d flow.toFluidFlow t))
    (hVelocitySpace : ∀ t, Differentiable ℝ (flow.velocity t)) :
    CauchyMomentumEquation d flow ↔ ConvectiveCauchyMomentumEquation d flow
```

2.17 Physlib.FluidDynamics.CauchyFlow.NavierStokes

Physlib/FluidDynamics/CauchyFlow/NavierStokes.lean on GitHub.

theorem navier_stokes_iff_convective_navier_stokes [source]

The conservative and convective Navier-Stokes forms are equivalent when the fields are differentiable enough for the product rules.

```
theorem navier_stokes_iff_convective_navier_stokes
    (d : ℕ) (flow : CauchyFlow d)
    (pressure shearViscosity secondViscosity : ScalarField d)
    (hRhoTime : ∀ t x, DifferentiableAt ℝ (flow.rho · x) t)
    (hVelocityTime : ∀ t x, DifferentiableAt ℝ (flow.velocity · x) t)
    (hMomentumDensity : ∀ t,
      Differentiable ℝ (FluidFlow.momentumDensity d flow.toFluidFlow t))
    (hVelocitySpace : ∀ t, Differentiable ℝ (flow.velocity t)) :
    NavierStokes d flow pressure shearViscosity secondViscosity ↔
      ConvectiveNavierStokes d flow pressure shearViscosity secondViscosity
```

2.18 Physlib.FluidDynamics.CauchyFlow.Newtonian

Physlib/FluidDynamics/CauchyFlow/Newtonian.lean on GitHub.

def IsNewtonian [source]

A Cauchy flow is Newtonian when its stress tensor has the Newtonian constitutive form.

```
def IsNewtonian
    (d : ℕ) (flow : CauchyFlow d)
    (pressure shearViscosity secondViscosity : ScalarField d) : Prop
```

2.19 Physlib.FluidDynamics.Euler.Basic

Physlib/FluidDynamics/Euler/Basic.lean on GitHub.

def Euler [source]

The conservative Euler equations: continuity, Cauchy momentum, and inviscid stress.

```
def Euler (d : ℕ) (flow : CauchyFlow d) (pressure : ScalarField d) : Prop
```

def ConvectiveEuler [source]

The convective Euler equations: continuity, convective Cauchy momentum, and inviscid stress.

```
def ConvectiveEuler (d : ℕ) (flow : CauchyFlow d) (pressure : ScalarField d) : Prop
```

theorem euler_iff_convectiveEuler [source]

The conservative and convective Euler forms are equivalent when the fields are differentiable enough for the product rules.

```
theorem euler_iff_convectiveEuler
    (d : ℕ) (flow : CauchyFlow d) (pressure : ScalarField d)
    (hRhoTime : ∀ t x, DifferentiableAt ℝ (flow.rho · x) t)
    (hVelocityTime : ∀ t x, DifferentiableAt ℝ (flow.velocity · x) t)
    (hMomentumDensity : ∀ t,
      Differentiable ℝ (FluidFlow.momentumDensity d flow.toFluidFlow t))
    (hVelocitySpace : ∀ t, Differentiable ℝ (flow.velocity t)) :
    Euler d flow pressure ↔ ConvectiveEuler d flow pressure
```

2.20 Physlib.FluidDynamics.FluidFlow.Basic

Physlib/FluidDynamics/FluidFlow/Basic.lean on GitHub.

structure FluidFlow [source]

The density and velocity fields of a fluid on `d`-dimensional space.

These fields are the independent data needed for kinematics and mass transport. Quantities such as mass flux, momentum density, and material acceleration are derived from them. Dynamic and thermodynamic fields are introduced only by extension structures whose statements need them.

```
structure FluidFlow (d : ℕ) where
  rho : MassDensity d
  velocity : VelocityField d
```

2.21 Physlib.FluidDynamics.FluidFlow.Continuity

Physlib/FluidDynamics/FluidFlow/Continuity.lean on GitHub.

def ClassicalContinuityEquation [source]

Classical conservation of mass in conservative form, `partial_t rho + div (rho u) = 0`.

The equation is asserted at points where the time derivative of `rho` and the spatial divergence of `rho u` are classical derivatives.

```
def ClassicalContinuityEquation (d : ℕ) (fluid : FluidFlow d) : Prop
```

def continuityResidual [source]

The scalar continuity-equation residual `partial_t rho + div (rho u)`.

```
noncomputable def continuityResidual (d : ℕ) (fluid : FluidFlow d) : Time → Space d → ℝ
```

def SmoothContinuityEquation [source]

A stronger continuity equation for globally differentiable fields.

This version records the first-order regularity needed by the classical continuity equation: the density is differentiable in time, the mass flux `rho u` is differentiable in space, and the continuity residual vanishes everywhere.

```
def SmoothContinuityEquation (d : ℕ) (fluid : FluidFlow d) : Prop
```

lemma classicalContinuityEquation_of_smoothContinuityEquation [source]

A smooth continuity equation satisfies the guarded classical continuity equation.

```
lemma classicalContinuityEquation_of_smoothContinuityEquation
    (d : ℕ) (fluid : FluidFlow d) :
    SmoothContinuityEquation d fluid → ClassicalContinuityEquation d fluid
```

2.22 Physlib.FluidDynamics.FluidFlow.Incompressible

Physlib/FluidDynamics/FluidFlow/Incompressible.lean on GitHub.

def incompressibilityResidual [source]

The incompressibility residual, given by the divergence of the velocity field.

```
noncomputable def incompressibilityResidual (d : ℕ) (fluid : FluidFlow d) :
    Time → Space d → ℝ
```

def ClassicalIncompressible [source]

A classical incompressible flow has divergence-free velocity at points where the velocity field is differentiable.

```
def ClassicalIncompressible (d : ℕ) (fluid : FluidFlow d) : Prop
```

def SmoothIncompressible [source]

A smooth incompressible flow has globally differentiable velocity and vanishing incompressibility residual everywhere.

```
def SmoothIncompressible (d : ℕ) (fluid : FluidFlow d) : Prop
```

lemma classicalIncompressible_of_smoothIncompressible [source]

A smooth incompressible flow is classically incompressible.

```
lemma classicalIncompressible_of_smoothIncompressible
    (d : ℕ) (fluid : FluidFlow d) :
    SmoothIncompressible d fluid → ClassicalIncompressible d fluid
```

2.23 Physlib.FluidDynamics.FluidFlow.Kinematics

Physlib/FluidDynamics/FluidFlow/Kinematics.lean on GitHub.

def materialDerivative [source]

The material derivative `D_t f = partial_t f + u · grad f` of a scalar field.

```
noncomputable def materialDerivative (d : ℕ) (fluid : FluidFlow d)
    (field : ScalarField d) : ScalarField d
```

def specificKineticEnergy [source]

The specific kinetic energy `|u|^2 / 2` of a fluid flow.

```
noncomputable def specificKineticEnergy (d : ℕ) (fluid : FluidFlow d) : ScalarField d
```

2.24 Physlib.FluidDynamics.FluidFlow.Momentum

Physlib/FluidDynamics/FluidFlow/Momentum.lean on GitHub.

def momentumDensity [source]

The momentum density `rho u`.

```
def momentumDensity (d : ℕ) (fluid : FluidFlow d) : MomentumDensityField d
```

def momentumFlux [source]

The convective momentum flux `rho u ⊗ u`.

```
def momentumFlux (d : ℕ) (fluid : FluidFlow d) : Time → Space d → Matrix (Fin d) (Fin d) ℝ
```

def convectiveTerm [source]

The nonlinear transport term `(u · ∇)u`.

```
noncomputable def convectiveTerm (d : ℕ) (fluid : FluidFlow d) : VectorField d
```

def materialAcceleration [source]

The material acceleration `∂ₜ u + (u · ∇)u`.

```
noncomputable def materialAcceleration (d : ℕ) (fluid : FluidFlow d) : VectorField d
```

lemma timeDeriv_smul_velocity [source]

Product rule for the time derivative of a scalar field times a velocity field.

```
lemma timeDeriv_smul_velocity (d : ℕ) (rhoAtPosition : Time → ℝ)
    (velocityAtPosition : Time → EuclideanSpace ℝ (Fin d)) (t : Time)
    (hRho : DifferentiableAt ℝ rhoAtPosition t)
    (hVelocity : DifferentiableAt ℝ velocityAtPosition t) :
    ∂ₜ (fun t' => rhoAtPosition t' • velocityAtPosition t') t =
      rhoAtPosition t • ∂ₜ velocityAtPosition t + ∂ₜ rhoAtPosition t • velocityAtPosition t
```

lemma timeDeriv_momentumDensity [source]

Product rule for the time derivative of the momentum density `rho u`.

```
lemma timeDeriv_momentumDensity (d : ℕ) (fluid : FluidFlow d)
    (t : Time) (x : Space d)
    (hRho : DifferentiableAt ℝ (fluid.rho · x) t)
    (hVelocity : DifferentiableAt ℝ (fluid.velocity · x) t) :
    ∂ₜ (momentumDensity d fluid · x) t =
      fluid.rho t x • ∂ₜ (fluid.velocity · x) t + ∂ₜ (fluid.rho · x) t • fluid.velocity t x
```

lemma spaceDeriv_momentumFlux_component [source]

Product rule for one spatial derivative of one component of `rho u ⊗ u`.

```
lemma spaceDeriv_momentumFlux_component (d : ℕ) (fluid : FluidFlow d)
    (t : Time) (x : Space d) (i j : Fin d)
    (hMomentumDensity : Differentiable ℝ (momentumDensity d fluid t))
    (hVelocity : Differentiable ℝ (fluid.velocity t)) :
    ∂[j] (fun x' => momentumFlux d fluid t x' i j) x =
      fluid.velocity t x i • ∂[j] (fun x' => momentumDensity d fluid t x' j) x +
      ∂[j] (fun x' => fluid.velocity t x' i) x • momentumDensity d fluid t x j
```

lemma matrixDiv_momentumFlux [source]

The matrix divergence of `rho u ⊗ u` split into continuity and convective parts.

```
lemma matrixDiv_momentumFlux (d : ℕ) (fluid : FluidFlow d)
    (t : Time) (x : Space d)
    (hMomentumDensity : Differentiable ℝ (momentumDensity d fluid t))
    (hVelocity : Differentiable ℝ (fluid.velocity t)) :
    matrixDiv d (momentumFlux d fluid t) x =
      (∇ · momentumDensity d fluid t) x • fluid.velocity t x +
        fluid.rho t x • convectiveTerm d fluid t x
```

lemma momentumTransport_eq_materialAcceleration_add_continuityResidual [source]

The conservative momentum transport terms equal mass density times material acceleration plus the continuity residual times velocity.

```
lemma momentumTransport_eq_materialAcceleration_add_continuityResidual
    (d : ℕ) (fluid : FluidFlow d)
    (t : Time) (x : Space d)
    (hRhoTime : DifferentiableAt ℝ (fluid.rho · x) t)
    (hVelocityTime : DifferentiableAt ℝ (fluid.velocity · x) t)
    (hMomentumDensity : Differentiable ℝ (momentumDensity d fluid t))
    (hVelocitySpace : Differentiable ℝ (fluid.velocity t)) :
    ∂ₜ (momentumDensity d fluid · x) t + matrixDiv d (momentumFlux d fluid t) x =
      fluid.rho t x • materialAcceleration d fluid t x +
        continuityResidual d fluid t x • fluid.velocity t x
```

2.25 Physlib.FluidDynamics.FluidFlow.Newtonian

Physlib/FluidDynamics/FluidFlow/Newtonian.lean on GitHub.

def velocityGradient [source]

The spatial velocity-gradient matrix, with entries `partial_j u_i`.

```
noncomputable def velocityGradient (d : ℕ) (flow : FluidFlow d) :
    Time → Space d → Matrix (Fin d) (Fin d) ℝ
```

def newtonianStressTensor [source]

The Newtonian stress tensor `-p I + mu (grad u + grad u^T) + lambda (div u) I`.

The scalar fields are pressure, shear viscosity, and second viscosity.

```
noncomputable def newtonianStressTensor (d : ℕ) (flow : FluidFlow d)
    (pressure shearViscosity secondViscosity : ScalarField d) : StressTensor d
```

2.26 Physlib.FluidDynamics.ThermodynamicCauchyFlow.Basic

Physlib/FluidDynamics/ThermodynamicCauchyFlow/Basic.lean on GitHub.

structure ThermodynamicCauchyFlow [source]

A Cauchy flow equipped with thermodynamic entropy and enthalpy fields.

These fields are independent of the kinematic and momentum-balance data unless additional thermodynamic constitutive laws are imposed. They are sufficient for the current isentropic and Bernoulli APIs; other thermodynamic fields belong in later extensions that require them.

```
structure ThermodynamicCauchyFlow (d : ℕ) extends CauchyFlow d where
  entropy : ScalarField d
  enthalpy : ScalarField d
```

2.27 Physlib.FluidDynamics.ThermodynamicCauchyFlow.Bernoulli

Physlib/FluidDynamics/ThermodynamicCauchyFlow/Bernoulli.lean on GitHub.

def bernoulliFunction [source]

The Bernoulli function `|u|^2 / 2 + h + Phi`.

```
noncomputable def bernoulliFunction
    (d : ℕ) (flow : ThermodynamicCauchyFlow d) (potential : Space d → ℝ) : ScalarField d
```

def LocalBernoulliLaw [source]

A local Bernoulli law: the Bernoulli function has zero spatial gradient.

```
def LocalBernoulliLaw
    (d : ℕ) (flow : ThermodynamicCauchyFlow d) (potential : Space d → ℝ) : Prop
```

def BernoulliLaw [source]

A global Bernoulli law: the Bernoulli function is spatially constant at each time.

```
def BernoulliLaw
    (d : ℕ) (flow : ThermodynamicCauchyFlow d) (potential : Space d → ℝ) : Prop
```

2.28 Physlib.FluidDynamics.ThermodynamicCauchyFlow.Isentropic

Physlib/FluidDynamics/ThermodynamicCauchyFlow/Isentropic.lean on GitHub.

def IsIsentropic [source]

A thermodynamic flow is isentropic when the entropy is materially conserved along the underlying fluid velocity field.

```
def IsIsentropic (d : ℕ) (flow : ThermodynamicCauchyFlow d) : Prop
```

2.29 Physlib.Mathematics.CrossProduct

Physlib/Mathematics/CrossProduct.lean on GitHub.

lemma cross_cross_self_apply [source]

The component form of the triple cross product `v ×₃ (w ×₃ v)`: by the `bac-cab` identity its `i`-th entry is `|v|² wᵢ − (v · w) vᵢ`, written with the explicit component sums `∑ k, (v k)²` and `∑ j, v j * w j`.

```
lemma cross_cross_self_apply (v w : Fin 3 → ℝ) (i : Fin 3) :
    (v ×₃ (w ×₃ v)) i = (∑ k, (v k) ^ 2) * w i - (∑ j, v j * w j) * v i
```

lemma dotProduct_cross_cross_self [source]

Contracting `w` with the triple cross product `v ×₃ (w ×₃ v)` gives `(w ×₃ v) ⬝ᵥ (w ×₃ v)` (over `ℝ`, the squared length `|w × v|²`), by two cyclic permutations of the scalar triple product.

```
lemma dotProduct_cross_cross_self {R : Type*} [CommRing R] (v w : Fin 3 → R) :
    w ⬝ᵥ (v ×₃ (w ×₃ v)) = (w ×₃ v) ⬝ᵥ (w ×₃ v)
```

lemma contDiff_cross_dotProduct_cross [source]

The squared length `(ω ×₃ v) ⬝ᵥ (ω ×₃ v)` of the cross product with a fixed vector `ω` is a smooth function of `v`.

```
lemma contDiff_cross_dotProduct_cross (ω : Fin 3 → ℝ) :
    ContDiff ℝ ⊤ fun v : Fin 3 → ℝ => (ω ×₃ v) ⬝ᵥ (ω ×₃ v)
```

2.30 Physlib.Mathematics.CrossProductMatrix

Physlib/Mathematics/CrossProductMatrix.lean on GitHub.

lemma crossProductMatrix_crossProductVee [source]

On skew-symmetric matrices the hat map is also a right inverse of the vee map: if A^T = -A then `[Aᵛ]ₓ = A`. Together with `crossProductVee_crossProductMatrix` this identifies `ℝ³` with the skew-symmetric `3 × 3` matrices `𝔰𝔬(3)`.

```
lemma crossProductMatrix_crossProductVee {A : Matrix (Fin 3) (Fin 3) ℝ} (hA : A^T = -A) :
    crossProductMatrix (crossProductVee A) = A
```

2.31 Physlib.Mathematics.Distribution.Basic

Physlib/Mathematics/Distribution/Basic.lean on GitHub.

def toTemperedDistribution [source]

The Mathlib tempered distribution associated to a Physlib distribution.

```
def toTemperedDistribution (u : E →d[ℂ] F) : 𝒮'(E, F)
```

lemma toTemperedDistribution_apply [source]

```
@[simp]
lemma toTemperedDistribution_apply (u : E →d[ℂ] F) (η : 𝒮(E, ℂ)) :
    u.toTemperedDistribution η = u η
```

lemma toTemperedDistribution_fourierTransform [source]

Conversion to Mathlib tempered distributions commutes with the Fourier transform.

```
@[simp]
lemma toTemperedDistribution_fourierTransform (u : E →d[ℂ] F) :
    (u.fourierTransform E F).toTemperedDistribution = ℱ u.toTemperedDistribution
```

def ofFiniteMeasure [source]

The scalar distribution associated to a finite measure, acting on a Schwartz map by integration.

```
def ofFiniteMeasure (μ : Measure E) [IsFiniteMeasure μ] : E →d[𝕜] 𝕜
```

lemma ofFiniteMeasure_apply [source]

```
@[simp]
lemma ofFiniteMeasure_apply (μ : Measure E) [IsFiniteMeasure μ] (η : 𝒮(E, 𝕜)) :
    ofFiniteMeasure 𝕜 μ η = ∫ x, η x ∂μ
```

lemma toTemperedDistribution_ofFiniteMeasure [source]

The finite-measure distribution agrees with Mathlib's tempered distribution associated to the same measure.

```
@[simp]
lemma toTemperedDistribution_ofFiniteMeasure (μ : Measure E) [IsFiniteMeasure μ] :
    (ofFiniteMeasure ℂ μ).toTemperedDistribution = μ.toTemperedDistribution
```

lemma lipschitzWith_integral_of_le [source]

```
private lemma lipschitzWith_integral_of_le {μ ρ : Measure E} (hμρ : μ ≤ ρ) :
    LipschitzWith 1 (fun f : Lp ℂ 1 ρ => ∫ x, f x ∂μ)
```

lemma integral_boundedContinuous_eq_of_forall_schwartz_integral_eq [source]

```
private lemma integral_boundedContinuous_eq_of_forall_schwartz_integral_eq
    [FiniteDimensional ℝ E] {μ ν : Measure E} [IsFiniteMeasure μ] [IsFiniteMeasure ν]
    (h : ∀ η : 𝓢(E, ℂ), ∫ x, η x ∂μ = ∫ x, η x ∂ν)
    (f : BoundedContinuousFunction E ℂ) :
    ∫ x, f x ∂μ = ∫ x, f x ∂ν
```

lemma measure_eq_of_forall_schwartz_integral_eq [source]

```
private lemma measure_eq_of_forall_schwartz_integral_eq
    {E : Type} [NormedAddCommGroup E] [NormedSpace ℝ E] [MeasurableSpace E]
    [BorelSpace E] [SecondCountableTopology E] [FiniteDimensional ℝ E]
    {μ ν : Measure E} [IsFiniteMeasure μ] [IsFiniteMeasure ν]
    (h : ∀ η : 𝓢(E, ℂ), ∫ x, η x ∂μ = ∫ x, η x ∂ν) :
    μ = ν
```

lemma ofFiniteMeasure_eq_iff [source]

The complex scalar distribution associated to a finite measure determines the measure.

```
lemma ofFiniteMeasure_eq_iff
    {E : Type} [NormedAddCommGroup E] [NormedSpace ℝ E] [MeasurableSpace E]
    [BorelSpace E] [SecondCountableTopology E] [FiniteDimensional ℝ E]
    {μ ν : Measure E} [IsFiniteMeasure μ] [IsFiniteMeasure ν] :
    ofFiniteMeasure ℂ μ = ofFiniteMeasure ℂ ν ↔ μ = ν
```

2.32 Physlib.Mathematics.InnerProductSpace.Gaussian

Physlib/Mathematics/InnerProductSpace/Gaussian.lean on GitHub.

lemma norm_iteratedFDeriv_innerSL_le_pow [source]

Crude bounds on the norms of iterated Fréchet derivatives of `innerSL ℝ` in the form required by `norm_iteratedFDeriv_comp_le`.

```
private lemma norm_iteratedFDeriv_innerSL_le_pow {n : ℕ} (hn : 1 ≤ n) :
    ‖iteratedFDeriv ℝ n (fun x : E ↦ innerSL ℝ x x) x‖ ≤ (2 + ‖x‖ ^ 2) ^ n
```

lemma pow_mul_exp_bddAbove [source]

```
private lemma pow_mul_exp_bddAbove {s : ℝ} (hs : 0 ≤ s) (n : ℕ) :
    ∃ C > 0, ∀ x ≥ 0, x ^ s * (2 + x) ^ n * rexp (-2⁻¹ * x) ≤ C
```

def realStdGaussian [source]

The (unnormalized) real-valued standard Gaussian `exp (-2⁻¹ * ‖x‖ ^ 2)` as a Schwartz map.

This definition is used to construct the general RCLike-valued Gaussian and the preferred version to use is `gaussian ℝ B x₀` (or `stdGaussian E ℝ` for mean zero, variance one).

```
def realStdGaussian : 𝓢(E, ℝ) where
```

def gaussian [source]

The (unnormalized) 𝕜-valued Gaussian `exp (-2⁻¹ * ‖B⁻¹ (x - x₀)‖ ^ 2)` as a Schwartz map.

```
def gaussian : 𝓢(E, 𝕜)
```

lemma gaussian_apply [source]

```
@[simp]
lemma gaussian_apply : gaussian 𝕜 B x₀ x = ofReal (rexp (-2⁻¹ * ‖B.symm (x - x₀)‖ ^ 2))
```

abbrev gaussian₀ [source]

The (unnormalized) 𝕜-valued Gaussian `exp (-2⁻¹ * ‖B⁻¹ x‖ ^ 2)` as a Schwartz map.

```
abbrev gaussian₀ : 𝓢(E, 𝕜)
```

abbrev stdGaussian [source]

The (unnormalized) 𝕜-valued standard Gaussian `exp (-2⁻¹ * ‖x‖ ^ 2)` as a Schwartz map.

```
abbrev stdGaussian : 𝓢(E, 𝕜)
```

2.33 Physlib.Mathematics.KroneckerDelta.Basic

Physlib/Mathematics/KroneckerDelta/Basic.lean on GitHub.

lemma kroneckerDelta_finSumFinEquiv [source]

The Kronecker delta is invariant under the component-index equivalence `finSumFinEquiv`.

```
lemma kroneckerDelta_finSumFinEquiv (a b : Fin 1 ⊕ Fin 3) :
    kroneckerDelta (finSumFinEquiv a) (finSumFinEquiv b) = kroneckerDelta a b
```

2.34 Physlib.Mathematics.KroneckerDelta.Contraction

Physlib/Mathematics/KroneckerDelta/Contraction.lean on GitHub.

lemma det_add_rankOne_aux [source]

Expanding the determinant of a rank-one row update over a finite set of rows. For `i ∈ s` the row `A i` is replaced by `A i + w i • b`; the other rows are untouched.

```
private lemma det_add_rankOne_aux {ι : Type*} [DecidableEq ι] [Fintype ι] {R : Type*}
    [CommRing R] (A : Matrix ι ι R) (w b : ι → R) (s : Finset ι) :
    (A + Matrix.of fun i j => (if i ∈ s then w i else 0) * b j).det
      = A.det + ∑ i ∈ s, w i * (A.updateRow i b).det
```

lemma det_add_rankOne [source]

***Rank-one determinant update** (the ring-general matrix determinant lemma for an outer product, valid even when `A` is singular). Adding the rank-one matrix `w ⊗ b` to `A` changes the determinant by `∑ i, w i * det (A.updateRow i b)`.*

Mathlib only provides this when `det A` is a unit (`Matrix.det_add_replicateCol_mul_replicateRow`); the singular case is needed here because Kronecker-delta matrices are typically singular.

```
private lemma det_add_rankOne {ι : Type*} [DecidableEq ι] [Fintype ι] {R : Type*}
    [CommRing R] (A : Matrix ι ι R) (w b : ι → R) :
    (A + Matrix.of fun i j => w i * b j).det = A.det + ∑ i, w i * (A.updateRow i b).det
```

lemma generalizedKroneckerDelta_mul [source]

The product of two Levi-Civita-type symbols is a generalized Kronecker delta: `δ^{μ}_{·} · δ^{ν}_{·} = δ^{μ}_{ν}`, where each single factor is a Kronecker matrix against the identity. This is the Lean form of `ε^{μ₁…μₙ} ε_{ν₁…νₙ} = δ^{μ₁…μₙ}_{ν₁…νₙ}`.

```
lemma generalizedKroneckerDelta_mul (μ ν : α → α) :
    generalizedKroneckerDelta μ id * generalizedKroneckerDelta ν id
      = generalizedKroneckerDelta μ ν
```

lemma generalizedKroneckerDelta_sum_snoc [source]

***Generalized Kronecker delta contraction.** Summing a `generalizedKroneckerDelta` over one shared index appended at the end lowers the rank by one and pulls out a factor of `card α - n`. This is the reusable combinatorial fact behind all epsilon-epsilon identities.*

```
lemma generalizedKroneckerDelta_sum_snoc {n : ℕ} (μ ν : Fin n → α) :
    ∑ a : α, generalizedKroneckerDelta (Fin.snoc μ a) (Fin.snoc ν a)
      = ((Fintype.card α : ℤ) - n) * generalizedKroneckerDelta μ ν
```

lemma sum_over_snoc [source]

Split a sum over `(k+1)`-tuples into the last entry and the initial `k`-tuple.

```
private lemma sum_over_snoc {X : Type*} [Fintype X] {M : Type*} [AddCommMonoid M] {k : ℕ}
    (F : (Fin (k + 1) → X) → M) :
    ∑ h : Fin (k + 1) → X, F h = ∑ h' : Fin k → X, ∑ c : X, F (Fin.snoc h' c)
```

lemma sum_generalizedKroneckerDelta_self [source]

***Full contraction.** Iterating the snoc contraction over all four indices: `∑_f δ^{f}_{f} = 4!`. Here `f` ranges over all maps `Fin 4 → Fin 4`.*

```
lemma sum_generalizedKroneckerDelta_self (k : ℕ) :
    ∑ h : Fin k → Fin 4, generalizedKroneckerDelta h h
      = ∏ j ∈ Finset.range k, ((4 : ℤ) - j)
```

lemma sum_generalizedKroneckerDelta_cons [source]

***Single contraction.** Contracting the last `k` of `k+1` index pairs leaves one free pair `σ, τ`, with the factorial factor `(4-1)(4-2)…`.*

```
lemma sum_generalizedKroneckerDelta_cons (σ τ : Fin 4) (k : ℕ) :
    ∑ h : Fin k → Fin 4,
      generalizedKroneckerDelta (Fin.cons σ h) (Fin.cons τ h)
    = (∏ j ∈ Finset.range k, ((3 : ℤ) - j)) * ((kroneckerDelta σ τ : ℕ) : ℤ)
```

lemma sum_generalizedKroneckerDelta_cons₂ [source]

***Double contraction.** Contracting the last `k` of `k+2` index pairs leaves two free pairs, with value a `2×2` generalized Kronecker delta times the factorial factor.*

```
lemma sum_generalizedKroneckerDelta_cons₂ (ρ σ τ ω : Fin 4) (k : ℕ) :
    ∑ h : Fin k → Fin 4,
      generalizedKroneckerDelta (Fin.cons ρ (Fin.cons σ h)) (Fin.cons τ (Fin.cons ω h))
    = (∏ j ∈ Finset.range k, ((2 : ℤ) - j))
      * generalizedKroneckerDelta ![ρ, σ] ![τ, ω]
```

lemma sum_generalizedKroneckerDelta_mul_self [source]

Symbol-level full contraction over `Fin 4 → Fin 4`.

```
lemma sum_generalizedKroneckerDelta_mul_self :
    ∑ g : Fin 4 → Fin 4,
    generalizedKroneckerDelta g id * generalizedKroneckerDelta g id = (24 : ℤ)
```

lemma sum_generalizedKroneckerDelta_mul_cons [source]

Symbol-level triple contraction, one free pair `σ, τ`.

```
lemma sum_generalizedKroneckerDelta_mul_cons (σ τ : Fin 4) :
    ∑ h : Fin 3 → Fin 4,
      generalizedKroneckerDelta (Fin.cons σ h) id
        * generalizedKroneckerDelta (Fin.cons τ h) id
    = 6 * ((kroneckerDelta σ τ : ℕ) : ℤ)
```

lemma sum_generalizedKroneckerDelta_mul_cons₂ [source]

Symbol-level double contraction, two free pairs.

```
lemma sum_generalizedKroneckerDelta_mul_cons₂ (ρ σ τ ω : Fin 4) :
    ∑ h : Fin 2 → Fin 4,
      generalizedKroneckerDelta (Fin.cons ρ (Fin.cons σ h)) id
        * generalizedKroneckerDelta (Fin.cons τ (Fin.cons ω h)) id
    = 2 * (((kroneckerDelta ρ τ : ℕ) : ℤ) * ((kroneckerDelta σ ω : ℕ) : ℤ)
      - ((kroneckerDelta ρ ω : ℕ) : ℤ) * ((kroneckerDelta σ τ : ℕ) : ℤ))
```

2.35 Physlib.Mathematics.OrthogonalMatrix

Physlib/Mathematics/OrthogonalMatrix.lean on GitHub.

lemma dotProduct_mulVec_orthogonal [source]

An orthogonal matrix preserves the dot product: if `Aᵀ * A = 1` then `(A *ᵥ v) ⬝ᵥ (A *ᵥ w) = v ⬝ᵥ w`. Taking `w = v` shows orthogonal matrices preserve squared lengths, `(A *ᵥ v) ⬝ᵥ (A *ᵥ v) = v ⬝ᵥ v`.

```
lemma dotProduct_mulVec_orthogonal {R : Type*} [CommRing R] {n : Type*} [Fintype n]
    [DecidableEq n] {A : Matrix n n R} (hA : Aᵀ * A = 1) (v w : n → R) :
    (A *ᵥ v) ⬝ᵥ (A *ᵥ w) = v ⬝ᵥ w
```

2.36 Physlib.Mathematics.Resolvent

Physlib/Mathematics/Resolvent.lean on GitHub.

lemma mem_resolventSet_of_im_ne_zero [source]

Every real number lies in the ℝ-resolvent set of a non-real complex number: `t - z` is invertible for all `t : ℝ`.

```
lemma mem_resolventSet_of_im_ne_zero (hz : z.im ≠ 0) (t : ℝ) : t ∈ resolventSet ℝ z
```

lemma resolventSet_eq_univ [source]

The ℝ-resolvent set of a non-real complex number is all of ℝ.

```
lemma resolventSet_eq_univ (hz : z.im ≠ 0) : resolventSet ℝ z = Set.univ
```

lemma norm_resolvent_le [source]

The resolvent is globally bounded by `|z.im|⁻¹`: the imaginary part of the denominator `t - z` is exactly `-z.im`.

```
lemma norm_resolvent_le (hz : z.im ≠ 0) (t : ℝ) : ‖resolvent z t‖ ≤ |z.im|⁻¹
```

lemma contDiff_resolvent [source]

The resolvent of a non-real complex number is smooth along ℝ.

```
@[fun_prop]
lemma contDiff_resolvent (hz : z.im ≠ 0) : ContDiff ℝ ∞ (resolvent (R := ℝ) z)
```

lemma iteratedDeriv_resolvent [source]

Closed form for the iterated derivatives of the resolvent: the `n`-th derivative is `(-1)ⁿ · n! · (resolvent z)ⁿ⁺¹`.

```
lemma iteratedDeriv_resolvent (hz : z.im ≠ 0) (n : ℕ) :
    iteratedDeriv n (resolvent (R := ℝ) z)
      = fun t ↦ (-1) ^ n * (n ! : ℂ) * resolvent z t ^ (n + 1)
```

lemma norm_iteratedDeriv_resolvent_le [source]

Every iterated derivative of the resolvent is globally bounded, explicitly by `n! · (|z.im| ^ (n + 1))⁻¹`.

```
lemma norm_iteratedDeriv_resolvent_le (hz : z.im ≠ 0) (n : ℕ) (t : ℝ) :
    ‖iteratedDeriv n (resolvent z) t‖ ≤ n ! * (|z.im| ^ (n + 1))⁻¹
```

lemma hasTemperateGrowth_resolvent [source]

The resolvent of a non-real complex number has temperate growth along ℝ.

```
@[fun_prop]
lemma hasTemperateGrowth_resolvent (hz : z.im ≠ 0) :
    Function.HasTemperateGrowth (resolvent (R := ℝ) z)
```

2.37 Physlib.Mathematics.SpecialFunctions.PhysHermite

Physlib/Mathematics/SpecialFunctions/PhysHermite.lean on GitHub.

def physHermite [source]

The physicist's Hermite polynomials are defined as polynomials over ℤ in `X` recursively with `physHermite 0 = 1` and

`physHermite (n + 1) = 2 • X * physHermite n - derivative (physHermite n)`.

This polynomial will often be cast as a function ℝ → ℝ by evaluating the polynomial at `X`.

```
noncomputable def physHermite : ℕ → Polynomial ℤ
  | 0 => 1
  | n + 1 => 2 • X * physHermite n - derivative (physHermite n)
```

lemma physHermite_succ [source]

The defining recursion `physHermite (n + 1) = (2x - d/dx) (physHermite n)`.

```
lemma physHermite_succ (n : ℕ) :
    physHermite (n + 1) = 2 • X * physHermite n - derivative (physHermite n)
```

lemma physHermite_eq_iterate [source]

The Rodrigues formula `physHermite n = (2x - d/dx)ⁿ 1`.

```
lemma physHermite_eq_iterate (n : ℕ) :
    physHermite n = (fun p => 2 * X * p - derivative p)^[n] 1
```

lemma physHermite_zero [source]

```
@[simp]
lemma physHermite_zero : physHermite 0 = C 1
```

lemma physHermite_one [source]

```
@[simp]
lemma physHermite_one : physHermite 1 = 2 * X
```

lemma derivative_physHermite_succ [source]

```
lemma derivative_physHermite_succ : (n : ℕ) →
    derivative (physHermite (n + 1)) = 2 * (n + 1) • physHermite n
  | 0 => by simp [physHermite_one]
  | n + 1 => by
    rw [physHermite_succ]
    simp only [derivative_physHermite_succ n, nsmul_eq_mul, Nat.cast_ofNat, Nat.cast_add,
      Nat.cast_one, derivative_sub, derivative_mul, derivative_ofNat, zero_mul, derivative_X,
      mul_one, zero_add, derivative_add, derivative_natCast, derivative_one, add_zero]
    simp only [physHermite_succ, nsmul_eq_mul]
    ring
```

lemma derivative_physHermite [source]

```
lemma derivative_physHermite : (n : ℕ) →
    derivative (physHermite n) = 2 * n • physHermite (n - 1)
  | 0 => by simp
  | n + 1 => by simp [derivative_physHermite_succ]
```

lemma physHermite_succ' [source]

```
lemma physHermite_succ' (n : ℕ) :
    physHermite (n + 1) = 2 • X * physHermite n - 2 * n • physHermite (n - 1)
```

lemma coeff_physHermite_succ_zero [source]

```
lemma coeff_physHermite_succ_zero (n : ℕ) :
    coeff (physHermite (n + 1)) 0 = - coeff (physHermite n) 1
```

lemma coeff_physHermite_succ_succ [source]

```
lemma coeff_physHermite_succ_succ (n k : ℕ) : coeff (physHermite (n + 1)) (k + 1) =
    2 * coeff (physHermite n) k - (k + 2) * coeff (physHermite n) (k + 2)
```

lemma coeff_physHermite_of_lt [source]

```
lemma coeff_physHermite_of_lt {n k : ℕ} (hnk : n < k) : coeff (physHermite n) k = 0
```

lemma coeff_physHermite_self [source]

```
@[simp]
lemma coeff_physHermite_self (n : ℕ) : coeff (physHermite n) n = 2 ^ n
```

lemma degree_physHermite [source]

```
@[simp]
lemma degree_physHermite (n : ℕ) : degree (physHermite n) = n
```

lemma natDegree_physHermite [source]

```
@[simp]
lemma natDegree_physHermite (n : ℕ) : (physHermite n).natDegree = n
```

lemma physHermite_leadingCoeff [source]

```
@[simp]
lemma physHermite_leadingCoeff (n : ℕ) : (physHermite n).leadingCoeff = 2 ^ n
```

lemma physHermite_ne_zero [source]

```
@[simp]
lemma physHermite_ne_zero (n : ℕ) : physHermite n ≠ 0
```

lemma iterate_derivative_physHermite_of_gt [source]

```
lemma iterate_derivative_physHermite_of_gt {n m : ℕ} (h : n < m) :
    derivative^[m] (physHermite n) = 0
```

lemma iterate_derivative_physHermite_self [source]

```
@[simp]
lemma iterate_derivative_physHermite_self (n : ℕ) :
    derivative^[n] (physHermite n) = C (n ! * 2 ^ n : ℤ)
```

abbrev realEval [source]

Cast an integer polynomial to a function ℝ → ℝ by evaluation of the indeterminant.

```
@[coe]
noncomputable abbrev realEval (p : Polynomial ℤ) : ℝ → ℝ
```

instance CoeFun (Polynomial ℤ) (fun _ ↦ ℝ → ℝ) [source]

```
noncomputable instance : CoeFun (Polynomial ℤ) (fun _ ↦ ℝ → ℝ)
```

lemma physHermite_zero_coe [source]

```
lemma physHermite_zero_coe : (physHermite 0 : ℝ → ℝ) = fun _ ↦ 1
```

lemma physHermite_succ_coe [source]

The defining recursion for `physHermite` as functions ℝ → ℝ (c.f. `physHermite_succ`).

```
lemma physHermite_succ_coe (n : ℕ) :
    (physHermite (n + 1) : ℝ → ℝ) =
      2 • (fun x => x) * (physHermite n : ℝ → ℝ) - deriv (physHermite n)
```

lemma physHermite_succ_apply [source]

```
lemma physHermite_succ_apply (n : ℕ) (x : ℝ) :
    physHermite (n + 1) x = 2 * x * physHermite n x - deriv (physHermite n) x
```

lemma physHermite_succ_coe' [source]

The two-term recursion for `physHermite` as functions `ℝ → ℝ` (c.f. `physHermite_succ'`).

```
lemma physHermite_succ_coe' (n : ℕ) :
    (physHermite (n + 1) : ℝ → ℝ) =
      2 • (fun x => x) * (physHermite n : ℝ → ℝ) - (2 * n) • (physHermite (n - 1) : ℝ → ℝ)
```

lemma physHermite_succ_apply' [source]

```
lemma physHermite_succ_apply' (n : ℕ) (x : ℝ) :
    physHermite (n + 1) x = 2 * x * physHermite n x - 2 * n * physHermite (n - 1) x
```

lemma deriv_physHermite [source]

```
lemma deriv_physHermite (n : ℕ) : deriv (physHermite n) = 2 * n * physHermite (n - 1)
```

lemma iterate_deriv_physHermite_eq_iterate_derivative [source]

```
lemma iterate_deriv_physHermite_eq_iterate_derivative (n m : ℕ) :
    deriv^[m] (physHermite n) = derivative^[m] (physHermite n)
```

lemma fderiv_physHermite [source]

```
lemma fderiv_physHermite (n : ℕ) (x : ℝ) :
    fderiv ℝ (physHermite n) x = (1 : ℝ →L[ℝ] ℝ).smulRight (deriv (physHermite n) x)
```

lemma physHermite_neg [source]

The `physHermite` polynomials are alternately even and odd.

```
@[simp]
lemma physHermite_neg (n : ℕ) (x : ℝ) : physHermite n (-x) = (-1) ^ n * physHermite n x
```

lemma physHermite_even [source]

```
lemma physHermite_even {n : ℕ} (hn : Even n) : Function.Even (physHermite n)
```

lemma physHermite_odd [source]

```
lemma physHermite_odd {n : ℕ} (hn : Odd n) : Function.Odd (physHermite n)
```

lemma physHermite_differentiable [source]

```
@[fun_prop]
lemma physHermite_differentiable (n : ℕ) : Differentiable ℝ (physHermite n)
```

lemma deriv_physHermite_differentiable [source]

```
@[fun_prop]
lemma deriv_physHermite_differentiable (n m : ℕ) : Differentiable ℝ (deriv^[m] (physHermite n))
```

lemma physHermite_continuous [source]

```
@[fun_prop]
lemma physHermite_continuous (n : ℕ) : Continuous (physHermite n)
```

lemma physHermite_contDiff [source]

```
@[fun_prop]
lemma physHermite_contDiff (n : ℕ) (m : WithTop ℕ∞) : ContDiff ℝ m (physHermite n)
```

lemma physHermite_hasTemperateGrowth [source]

```
@[fun_prop]
lemma physHermite_hasTemperateGrowth (n : ℕ) : HasTemperateGrowth (physHermite n)
```

lemma deriv_gaussian_eq_physHermite_mul_gaussian [source]

```
lemma deriv_gaussian_eq_physHermite_mul_gaussian (n : ℕ) (x : ℝ) :
    deriv^[n] (fun y => Real.exp (- y ^ 2)) x =
    (-1 : ℝ) ^ n * physHermite n x * Real.exp (- x ^ 2)
```

lemma physHermite_eq_deriv_gaussian [source]

```
lemma physHermite_eq_deriv_gaussian (n : ℕ) (x : ℝ) :
    physHermite n x = (-1 : ℝ) ^ n * deriv^[n]
    (fun y => Real.exp (- y ^ 2)) x / Real.exp (- x ^ 2)
```

lemma physHermite_eq_deriv_gaussian' [source]

```
lemma physHermite_eq_deriv_gaussian' (n : ℕ) (x : ℝ) :
    physHermite n x = (-1 : ℝ) ^ n * deriv^[n] (fun y => Real.exp (- y ^ 2)) x *
    Real.exp (x ^ 2)
```

lemma guassian_integrable_polynomial_cons [source]

```
@[fun_prop]
lemma guassian_integrable_polynomial_cons {b c : ℝ} (hb : 0 < b) (P : Polynomial ℤ) :
    MeasureTheory.Integrable fun x : ℝ => (P.aeval (c * x)) * Real.exp (-b * x ^ 2)
```

lemma guassian_integrable_polynomial [source]

```
@[fun_prop]
lemma guassian_integrable_polynomial {b : ℝ} (hb : 0 < b) (P : Polynomial ℤ) :
    MeasureTheory.Integrable fun x : ℝ => (P.aeval x) * Real.exp (-b * x ^ 2)
```

lemma physHermite_gaussian_integrable [source]

```
@[fun_prop]
lemma physHermite_gaussian_integrable (n p m : ℕ) :
    MeasureTheory.Integrable (deriv^[m] (physHermite p) * deriv^[n] fun x => Real.exp (-x ^ 2))
    MeasureTheory.volume
```

lemma integral_physHermite_mul_physHermite_eq_integral_deriv_exp [source]

```
lemma integral_physHermite_mul_physHermite_eq_integral_deriv_exp (n m : ℕ) :
    ∫ x : ℝ, (physHermite n x * physHermite m x) * Real.exp (-x ^ 2) =
    (-1 : ℝ) ^ m * ∫ x : ℝ, (physHermite n x * (deriv^[m] fun x => Real.exp (-x ^ 2)) x)
```

lemma integral_physHermite_mul_physHermite_eq_integral_deriv_inductive [source]

```
lemma integral_physHermite_mul_physHermite_eq_integral_deriv_inductive (n m : ℕ) :
    (p : ℕ) → (hpm : p ≤ m) →
    ∫ x : ℝ, (physHermite n x * physHermite m x) * Real.exp (- x ^ 2) =
    (-1 : ℝ) ^ (m - p) * ∫ x : ℝ, (deriv^[p] (physHermite n) x *
    (deriv^[m - p] fun x => Real.exp (-x ^ 2)) x)
  | 0, h => integral_physHermite_mul_physHermite_eq_integral_deriv_exp n m
  | p + 1, h => by
    rw [integral_physHermite_mul_physHermite_eq_integral_deriv_inductive n m p (by omega),
      show m - p = m - (p + 1) + 1 by omega, Function.iterate_succ_apply', pow_succ, mul_assoc,
      Function.iterate_succ_apply']
    congr
    have hl : ∫ (x : ℝ), deriv^[p] (physHermite n) x *
        deriv (deriv^[m - (p + 1)] fun x => Real.exp (-x ^ 2)) x =
        - ∫ (x : ℝ), deriv (deriv^[p] (physHermite n)) x *
        deriv^[m - (p + 1)] (fun x => Real.exp (-x ^ 2)) x
```

lemma integral_physHermite_mul_physHermite_eq_integral_deriv [source]

```
lemma integral_physHermite_mul_physHermite_eq_integral_deriv (n m : ℕ) :
    ∫ x : ℝ, (physHermite n x * physHermite m x) * Real.exp (- x ^ 2) =
    ∫ x : ℝ, (deriv^[m] (physHermite n) x * (Real.exp (-x ^ 2)))
```

lemma physHermite_orthogonal_lt [source]

```
lemma physHermite_orthogonal_lt {n m : ℕ} (hnm : n < m) :
    ∫ x : ℝ, (physHermite n x * physHermite m x) * Real.exp (- x ^ 2) = 0
```

theorem physHermite_orthogonal [source]

```
theorem physHermite_orthogonal {n m : ℕ} (hnm : n ≠ m) :
    ∫ x : ℝ, (physHermite n x * physHermite m x) * Real.exp (- x ^ 2) = 0
```

lemma physHermite_orthogonal_cons [source]

```
lemma physHermite_orthogonal_cons {n m : ℕ} (hnm : n ≠ m) (c : ℝ) :
    ∫ x : ℝ, (physHermite n (c * x) * physHermite m (c * x)) *
    Real.exp (- c ^ 2 * x ^ 2) = 0
```

theorem physHermite_norm [source]

```
theorem physHermite_norm (n : ℕ) :
    ∫ x : ℝ, (physHermite n x * physHermite n x) * Real.exp (- x ^ 2) =
    ↑n ! * 2 ^ n * √Real.pi
```

lemma physHermite_norm_cons [source]

```
lemma physHermite_norm_cons (n : ℕ) (c : ℝ) :
    ∫ x : ℝ, (physHermite n (c * x) * physHermite n (c * x)) * Real.exp (- c ^ 2 * x ^ 2) =
    |c⁻¹| • (↑n ! * 2 ^ n * √Real.pi)
```

lemma polynomial_mem_physHermite_span_induction [source]

```
lemma polynomial_mem_physHermite_span_induction (P : Polynomial ℤ) : (n : ℕ) →
    (hn : P.natDegree = n) →
    (P : ℝ → ℝ) ∈ Submodule.span ℝ (Set.range (fun n => (physHermite n : ℝ → ℝ)))
  | 0, h => by
    obtain ⟨x, rfl⟩
```

lemma polynomial_mem_physHermite_span [source]

```
lemma polynomial_mem_physHermite_span (P : Polynomial ℤ) :
    (P : ℝ → ℝ) ∈ Submodule.span ℝ (Set.range (fun n => (physHermite n : ℝ → ℝ)))
```

lemma cos_mem_physHermite_span_topologicalClosure [source]

```
lemma cos_mem_physHermite_span_topologicalClosure (c : ℝ) :
    (fun (x : ℝ) => Real.cos (c * x)) ∈
    (Submodule.span ℝ (Set.range (fun n => (physHermite n : ℝ → ℝ)))).topologicalClosure
```

2.38 Physlib.Particles.StandardModel.Basic

Physlib/Particles/StandardModel/Basic.lean on GitHub.

lemma gaugeGroupℤ₆SU3OfRoot_eq_mul_id [source]

```
lemma gaugeGroupℤ₆SU3OfRoot_eq_mul_id (α : rootsOfUnity 6 ℂ) :
    (gaugeGroupℤ₆SU3OfRoot α).1 = ((α : ℂˣ) : ℂ) ^ 2 • 1
```

lemma gaugeGroupℤ₆SU3OfRoot_toEuclideanLin_apply [source]

```
lemma gaugeGroupℤ₆SU3OfRoot_toEuclideanLin_apply (α : rootsOfUnity 6 ℂ)
    (v : EuclideanSpace ℂ (Fin 3)) :
    (gaugeGroupℤ₆SU3OfRoot α).1.toEuclideanLin v = ((α : ℂˣ) : ℂ) ^ 2 • v
```

lemma gaugeGroupℤ₆SU2OfRoot_eq_mul_id [source]

```
lemma gaugeGroupℤ₆SU2OfRoot_eq_mul_id (α : rootsOfUnity 6 ℂ) :
    (gaugeGroupℤ₆SU2OfRoot α).1 = star ((α : ℂˣ) : ℂ) ^ 3 • 1
```

lemma gaugeGroupℤ₆SU2OfRoot_toEuclideanLin_apply [source]

```
lemma gaugeGroupℤ₆SU2OfRoot_toEuclideanLin_apply (α : rootsOfUnity 6 ℂ)
    (v : EuclideanSpace ℂ (Fin 2)) :
    (gaugeGroupℤ₆SU2OfRoot α).1.toEuclideanLin v = star ((α : ℂˣ) : ℂ) ^ 3 • v
```

lemma subgroup_le_subgroup_ℤ₆ [source]

```
lemma subgroup_le_subgroup_ℤ₆ (q : GaugeGroupQuot) : subgroup q ≤ gaugeGroupℤ₆SubGroup
```

2.39 Physlib.Particles.StandardModel.Fermions.DownSinglet

Physlib/Particles/StandardModel/Fermions/DownSinglet.lean on GitHub.

structure DownSinglet [source]

The target vector space of one Standard Model down-type singlet quark. It carries the `(3, 1)_{-2}` representation of the gauge group.

```
@[ext]
structure DownSinglet where
  val : Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3)
```

def valEquiv [source]

Identifies a down-type singlet with its underlying tensor-product value.

```
def valEquiv : DownSinglet ≃ Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) where
```

instance AddCommGroup DownSinglet [source]

```
instance : AddCommGroup DownSinglet
```

instance Module ℂ DownSinglet [source]

```
instance : Module ℂ DownSinglet
```

def valLinEquiv [source]

The linear identification with the underlying tensor product.

```
def valLinEquiv : DownSinglet ≃ₗ[ℂ]
    Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) where
```

lemma valLinEquiv_apply [source]

```
@[simp]
lemma valLinEquiv_apply (d : DownSinglet) : valLinEquiv d = d.val
```

lemma valLinEquiv_symm_apply [source]

```
lemma valLinEquiv_symm_apply
    (m : Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3)) :
    valLinEquiv.symm m = ⟨m⟩
```

lemma val_add [source]

```
@[simp]
lemma val_add (d₁ d₂ : DownSinglet) : (d₁ + d₂).val = d₁.val + d₂.val
```

lemma val_smul [source]

```
@[simp]
lemma val_smul (r : ℂ) (d : DownSinglet) : (r • d).val = r • d.val
```

def repLorentzGroup [source]

The right-handed Lorentz representation on down-type singlet quarks.

```
noncomputable def repLorentzGroup : Representation ℂ (SL(2,ℂ)) DownSinglet where
```

def repGaugeGroupI [source]

The `(3, 1)_{-2}` action of the unquotiented Standard Model gauge group.

```
noncomputable def repGaugeGroupI : Representation ℂ GaugeGroupI DownSinglet where
```

lemma repGaugeGroupI_tmul [source]

The gauge action on a pure spinor–colour tensor.

```
lemma repGaugeGroupI_tmul (g : GaugeGroupI) (ψ : Fermion.RightHandedWeyl)
    (v : EuclideanSpace ℂ (Fin 3)) :
    repGaugeGroupI g ⟨ψ ⊗ₜ v⟩ =
      ⟨(star g.toU1.1 ^ 2) • ψ ⊗ₜ g.toSU3.1.toEuclideanLin v⟩
```

lemma repGaugeGroupI_tmul_basis_eq_sum [source]

Expands the gauge action in the spinor–colour basis.

```
lemma repGaugeGroupI_tmul_basis_eq_sum (g : GaugeGroupI) (k : Fin 2) (i : Fin 3) :
    repGaugeGroupI g
      ⟨RightHandedWeyl.basis k ⊗ₜ[ℂ] EuclideanSpace.basisFun (Fin 3) ℂ i⟩ =
      ∑ i' : Fin 3, (star g.toU1.1 ^ 2 * g.toSU3.1 i' i) •
        ((⟨RightHandedWeyl.basis k ⊗ₜ[ℂ]
          EuclideanSpace.basisFun (Fin 3) ℂ i'⟩ : DownSinglet)
```

lemma repGaugeGroupI_eq_iff_mul_eq [source]

Two gauge elements induce the same action exactly when their hypercharge–colour coefficients agree.

```
lemma repGaugeGroupI_eq_iff_mul_eq {g₁ g₂ : GaugeGroupI} :
    repGaugeGroupI g₁ = repGaugeGroupI g₂ ↔ ∀ i i',
      star g₁.toU1.1 ^ 2 * g₁.toSU3.1 i' i =
        star g₂.toU1.1 ^ 2 * g₂.toSU3.1 i' i
```

lemma mem_repGaugeGroupI_ker_iff_eq [source]

Characterizes the full-group elements acting trivially on the down-type singlet.

```
lemma mem_repGaugeGroupI_ker_iff_eq {g : GaugeGroupI} :
    g ∈ repGaugeGroupI.ker ↔ ∃ a : ℂ,
      g.toSU3.1 = a • 1 ∧ a * star g.toU1.1 ^ 2 = 1
```

lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI [source]

The central $\mathbb{Z}_6$ subgroup acts trivially on `(3, 1)_{-2}`.

```
lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI :
    GaugeGroupQuot.subgroup .ℤ₆ ≤ repGaugeGroupI.ker
```

lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI [source]

Every supported quotient subgroup acts trivially on the down-type singlet.

```
lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI (Q : GaugeGroupQuot) :
    Q.subgroup ≤ repGaugeGroupI.ker
```

def repGaugeGroup [source]

The `(3, 1)_{-2}` representation for every supported global form of the Standard Model gauge group.

```
noncomputable def repGaugeGroup : (Q : GaugeGroupQuot) →
    Representation ℂ (GaugeGroup Q) DownSinglet
  | .I => repGaugeGroupI
  | .ℤ₆ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₆)
  | .ℤ₂ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₂)
  | .ℤ₃ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₃)
```

2.40 Physlib.Particles.StandardModel.Fermions.LeptonDoublet

Physlib/Particles/StandardModel/Fermions/LeptonDoublet.lean on GitHub.

structure LeptonDoublet [source]

The target vector space of one Standard Model lepton doublet. It carries the `(1, 2)_{-3}` representation of the gauge group.

```
@[ext]
structure LeptonDoublet where
  val : Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 2)
```

def valEquiv [source]

Identifies a lepton doublet with its underlying tensor-product value.

```
def valEquiv : LeptonDoublet ≃ Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 2) where
```

instance AddCommGroup LeptonDoublet [source]

```
instance : AddCommGroup LeptonDoublet
```

instance Module ℂ LeptonDoublet [source]

```
instance : Module ℂ LeptonDoublet
```

def valLinEquiv [source]

The linear identification with the underlying tensor product.

```
def valLinEquiv : LeptonDoublet ≃ₗ[ℂ]
    Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 2) where
```

lemma valLinEquiv_apply [source]

```
@[simp]
lemma valLinEquiv_apply (l : LeptonDoublet) : valLinEquiv l = l.val
```

lemma valLinEquiv_symm_apply [source]

```
lemma valLinEquiv_symm_apply
    (m : Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 2)) :
    valLinEquiv.symm m = ⟨m⟩
```

lemma val_add [source]

```
@[simp]
lemma val_add (l₁ l₂ : LeptonDoublet) : (l₁ + l₂).val = l₁.val + l₂.val
```

lemma val_smul [source]

```
@[simp]
lemma val_smul (r : ℂ) (l : LeptonDoublet) : (r • l).val = r • l.val
```

def repLorentzGroup [source]

The left-handed Lorentz representation on lepton doublets.

```
noncomputable def repLorentzGroup : Representation ℂ (SL(2,ℂ)) LeptonDoublet where
```

def repGaugeGroupI [source]

The `(1, 2)_{-3}` action of the unquotiented Standard Model gauge group.

```
noncomputable def repGaugeGroupI : Representation ℂ GaugeGroupI LeptonDoublet where
```

lemma repGaugeGroupI_tmul [source]

The gauge action on a pure spinor–weak tensor.

```
lemma repGaugeGroupI_tmul (g : GaugeGroupI) (v : Fermion.LeftHandedWeyl)
    (w : EuclideanSpace ℂ (Fin 2)) :
    repGaugeGroupI g ⟨v ⊗ₜ w⟩ =
      ⟨((star g.toU1.1 ^ 3) • v ⊗ₜ (g.toSU2.1.toEuclideanLin w))
```

lemma repGaugeGroupI_tmul_basis_eq_sum [source]

Expands the gauge action in the spinor–weak basis.

```
lemma repGaugeGroupI_tmul_basis_eq_sum (g : GaugeGroupI) (k j : Fin 2) :
    repGaugeGroupI g ⟨LeftHandedWeyl.basis k ⊗ₜ[ℂ]
      EuclideanSpace.basisFun (Fin 2) ℂ j⟩ =
      ∑ j' : Fin 2, (star g.toU1.1 ^ 3 * g.toSU2.1 j' j)
      • ⟨(LeftHandedWeyl.basis k ⊗ₜ[ℂ]
          EuclideanSpace.basisFun (Fin 2) ℂ j') : LeptonDoublet⟩
```

lemma repGaugeGroupI_eq_iff_mul_eq [source]

Two gauge elements induce the same action exactly when their weak-basis coefficients agree.

```
lemma repGaugeGroupI_eq_iff_mul_eq {g₁ g₂ : GaugeGroupI} :
    repGaugeGroupI g₁ = repGaugeGroupI g₂ ↔ ∀ j j',
    star g₁.toU1.1 ^ 3 * g₁.toSU2.1 j' j =
      star g₂.toU1.1 ^ 3 * g₂.toSU2.1 j' j
```

lemma mem_repGaugeGroupI_ker_iff_eq [source]

Characterizes the full-group elements acting trivially on the lepton doublet.

```
lemma mem_repGaugeGroupI_ker_iff_eq {g : GaugeGroupI} :
    g ∈ repGaugeGroupI.ker ↔ ∃ a : ℂ, g.toSU2.1 = a • 1 ∧
      a * star g.toU1.1 ^ 3 = 1
```

lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI [source]

The central $\mathbb{Z}_6$ subgroup acts trivially on `(1, 2)_{-3}`.

```
lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI :
    GaugeGroupQuot.subgroup .ℤ₆ ≤ repGaugeGroupI.ker
```

lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI [source]

Every supported quotient subgroup acts trivially on the lepton doublet.

```
lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI (Q : GaugeGroupQuot) :
    Q.subgroup ≤ repGaugeGroupI.ker
```

def repGaugeGroup [source]

The `(1, 2)_{-3}` representation for every supported global form of the Standard Model gauge group.

```
noncomputable def repGaugeGroup : (Q : GaugeGroupQuot) →
    Representation ℂ (GaugeGroup Q) LeptonDoublet
  | .I => repGaugeGroupI
  | .ℤ₆ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₆)
  | .ℤ₂ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₂)
  | .ℤ₃ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₃)
```

2.41 `Physlib.Particles.StandardModel.Fermions.LeptonSinglet`

Physlib/Particles/StandardModel/Fermions/LeptonSinglet.lean on GitHub.

`structure LeptonSinglet` [source]

The target vector space of one Standard Model charged-lepton singlet. It carries the `(1, 1)_{-6}` representation of the gauge group.

```
@[ext]
structure LeptonSinglet where
  val : Fermion.RightHandedWeyl
```

`def valEquiv` [source]

Identifies a charged-lepton singlet with its underlying Weyl spinor.

```
def valEquiv : LeptonSinglet ≃ Fermion.RightHandedWeyl where
```

`instance AddCommGroup LeptonSinglet` [source]

```
instance : AddCommGroup LeptonSinglet
```

`instance Module ℂ LeptonSinglet` [source]

```
instance : Module ℂ LeptonSinglet
```

`def valLinEquiv` [source]

The linear identification with the underlying Weyl-spinor space.

```
def valLinEquiv : LeptonSinglet ≃ₗ[ℂ] Fermion.RightHandedWeyl where
```

`lemma valLinEquiv_apply` [source]

```
@[simp]
lemma valLinEquiv_apply (l : LeptonSinglet) : valLinEquiv l = l.val
```

`lemma valLinEquiv_symm_apply` [source]

```
lemma valLinEquiv_symm_apply (m : Fermion.RightHandedWeyl) :
    valLinEquiv.symm m = ⟨m⟩
```

lemma val_add [source]

```
@[simp]
lemma val_add (l₁ l₂ : LeptonSinglet) : (l₁ + l₂).val = l₁.val + l₂.val
```

lemma val_smul [source]

```
@[simp]
lemma val_smul (r : ℂ) (l : LeptonSinglet) : (r • l).val = r • l.val
```

def repLorentzGroup [source]

The right-handed Lorentz representation on charged-lepton singlets.

```
noncomputable def repLorentzGroup : Representation ℂ (SL(2,ℂ)) LeptonSinglet where
```

def repGaugeGroupI [source]

The `(1, 1)_{-6}` action of the unquotiented Standard Model gauge group.

```
noncomputable def repGaugeGroupI : Representation ℂ GaugeGroupI LeptonSinglet where
```

lemma repGaugeGroupI_apply [source]

The gauge group acts on a charged-lepton singlet by the hypercharge scalar alone.

```
lemma repGaugeGroupI_apply (g : GaugeGroupI) (ψ : Fermion.RightHandedWeyl) :
    repGaugeGroupI g ⟨ψ⟩ = ⟨(star g.toU1.1 ^ 6) • ψ⟩
```

lemma repGaugeGroupI_basis [source]

The gauge action is diagonal in the standard Weyl basis.

```
lemma repGaugeGroupI_basis (g : GaugeGroupI) (k : Fin 2) :
    repGaugeGroupI g ⟨RightHandedWeyl.basis k⟩ =
      (star g.toU1.1 ^ 6) • ((RightHandedWeyl.basis k) : LeptonSinglet)
```

lemma repGaugeGroupI_eq_iff [source]

Two gauge elements induce the same action exactly when their hypercharge scalars agree.

```
lemma repGaugeGroupI_eq_iff {g₁ g₂ : GaugeGroupI} :
    repGaugeGroupI g₁ = repGaugeGroupI g₂ ↔
      star g₁.toU1.1 ^ 6 = star g₂.toU1.1 ^ 6
```

lemma mem_repGaugeGroupI_ker_iff_eq [source]

Characterizes the full-group elements acting trivially on the charged-lepton singlet.

```
lemma mem_repGaugeGroupI_ker_iff_eq {g : GaugeGroupI} :
    g ∈ repGaugeGroupI.ker ↔ star g.toU1.1 ^ 6 = 1
```

lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI [source]

The central ℤ₆ subgroup acts trivially on `(1, 1)_{-6}`.

```
lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI :
    GaugeGroupQuot.subgroup .ℤ₆ ≤ repGaugeGroupI.ker
```

lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI [source]

Every supported quotient subgroup acts trivially on the charged-lepton singlet.

```
lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI (Q : GaugeGroupQuot) :
    Q.subgroup ≤ repGaugeGroupI.ker
```

def repGaugeGroup [source]

The `(1, 1)_{-6}` representation for every supported global form of the Standard Model gauge group.

```
noncomputable def repGaugeGroup : (Q : GaugeGroupQuot) →
    Representation ℂ (GaugeGroup Q) LeptonSinglet
  | .I => repGaugeGroupI
  | .ℤ₆ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₆)
  | .ℤ₂ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₂)
  | .ℤ₃ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₃)
```

2.42 Physlib.Particles.StandardModel.Fermions.QuarkDoublet

Physlib/Particles/StandardModel/Fermions/QuarkDoublet.lean on GitHub.

structure QuarkDoublet [source]

The vector space of a quark field in the Standard Model. These live in the `(3, 2)_{1}` representation of the gauge group.

```
@[ext]
structure QuarkDoublet where
  val : Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) ⊗[ℂ] EuclideanSpace ℂ (Fin 2)
```

def valEquiv [source]

The linear equivalence between `QuarkDoublet` and its underlying tensor product space.

```
def valEquiv : QuarkDoublet ≃
    Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) ⊗[ℂ] EuclideanSpace ℂ (Fin 2) where
```

instance AddCommGroup QuarkDoublet [source]

```
instance : AddCommGroup QuarkDoublet
```

instance Module ℂ QuarkDoublet [source]

```
instance : Module ℂ QuarkDoublet
```

def valLinEquiv [source]

The linear equivalence between `QuarkDoublet` and its underlying tensor product space.

```
def valLinEquiv : QuarkDoublet ≃ₗ[ℂ]
    Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) ⊗[ℂ] EuclideanSpace ℂ (Fin 2) where
```

lemma valLinEquiv_apply [source]

```
@[simp]
lemma valLinEquiv_apply (q : QuarkDoublet) : valLinEquiv q = q.val
```

lemma valLinEquiv_symm_apply [source]

```
lemma valLinEquiv_symm_apply
    (m : Fermion.LeftHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) ⊗[ℂ] EuclideanSpace ℂ (Fin 2)) :
    valLinEquiv.symm m = ⟨m⟩
```

lemma val_add [source]

```
@[simp]
lemma val_add (q1 q2 : QuarkDoublet) : (q1 + q2).val = q1.val + q2.val
```

lemma val_smul [source]

```
@[simp]
lemma val_smul (r : ℂ) (q : QuarkDoublet) : (r • q).val = r • q.val
```

def repLorentzGroup [source]

The representation of the Lorentz group on the space of quark fields.

```
noncomputable def repLorentzGroup : Representation ℂ (SL(2,ℂ)) QuarkDoublet where
```

def repGaugeGroupI [source]

The action of the full Standard Model gauge group on quark fields.

```
noncomputable def repGaugeGroupI : Representation ℂ GaugeGroupI QuarkDoublet where
```

lemma repGaugeGroupI_tmul [source]

```
lemma repGaugeGroupI_tmul (g : GaugeGroupI) (ψ : Fermion.LeftHandedWeyl)
    (v : EuclideanSpace ℂ (Fin 3)) (w : EuclideanSpace ℂ (Fin 2)) :
    repGaugeGroupI g (ψ ⊗ₜ v ⊗ₜ w) = ⟨g.toU1 • ψ ⊗ₜ (g.toSU3.1.toEuclideanLin v) ⊗ₜ
      (g.toSU2.1.toEuclideanLin w)⟩
```

lemma repGaugeGroupI_tmul_basis_eq_sum [source]

The action of the full gauge group on a tensor product of basis elements, expanded as a sum over the columns of the `SU(3)` and `SU(2)` matrices.

```
lemma repGaugeGroupI_tmul_basis_eq_sum (g : GaugeGroupI) (k : Fin 2) (i : Fin 3) (j : Fin 2) :
    repGaugeGroupI g (LeftHandedWeyl.basis k ⊗ₜ[ℂ] EuclideanSpace.basisFun (Fin 3) ℂ i
      ⊗ₜ[ℂ] EuclideanSpace.basisFun (Fin 2) ℂ j) =
      ∑ i' : Fin 3, ∑ j' : Fin 2, (g.toU1.1 * g.toSU3.1 i' i * g.toSU2.1 j' j)
      • ((LeftHandedWeyl.basis k ⊗ₜ[ℂ] EuclideanSpace.basisFun (Fin 3) ℂ i'
          ⊗ₜ[ℂ] EuclideanSpace.basisFun (Fin 2) ℂ j') : QuarkDoublet)
```

lemma repGaugeGroupI_eq_iff_mul_eq [source]

```
lemma repGaugeGroupI_eq_iff_mul_eq {g1 g2 : GaugeGroupI} :
    repGaugeGroupI g1 = repGaugeGroupI g2 ↔ ∀ i i' j j',
    g1.toU1.1 * g1.toSU3.1 i' i * g1.toSU2.1 j' j =
    g2.toU1.1 * g2.toSU3.1 i' i * g2.toSU2.1 j' j
```

lemma mem_repGaugeGroupI_ker_iff_eq [source]

```
lemma mem_repGaugeGroupI_ker_iff_eq {g : GaugeGroupI} :
    g ∈ repGaugeGroupI.ker ↔ ∃ a b : ℂ, g.toSU2.1 = a • 1 ∧ g.toSU3.1 = b • 1 ∧
      a * b * g.toU1.1 = 1
```

lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI [source]

```
lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI :
    GaugeGroupQuot.subgroup .ℤ₆ ≤ repGaugeGroupI.ker
```

lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI [source]

```
lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI (Q : GaugeGroupQuot) :
    Q.subgroup ≤ repGaugeGroupI.ker
```

def repGaugeGroup [source]

The action of the Standard Model gauge group, potentially quotiented by a discrete factor on quark fields.

```
noncomputable def repGaugeGroup : (Q : GaugeGroupQuot) →
    Representation ℂ (GaugeGroup Q) QuarkDoublet
  | .I => repGaugeGroupI
  | .ℤ₆ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₆)
  | .ℤ₂ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₂)
  | .ℤ₃ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₃)
```

2.43 Physlib.Particles.StandardModel.Fermions.UpSinglet

Physlib/Particles/StandardModel/Fermions/UpSinglet.lean on GitHub.

structure UpSinglet [source]

The vector space of an up-type singlet quark field in the Standard Model. These live in the (3, 1)_{4} representation of the gauge group.

```
@[ext]
structure UpSinglet where
```

```
  val : Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3)
```

def valEquiv [source]

The linear equivalence between `UpSinglet` and its underlying tensor product space.

```
def valEquiv : UpSinglet ≃ Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) where
```

instance AddCommGroup UpSinglet [source]

```
instance : AddCommGroup UpSinglet
```

instance Module ℂ UpSinglet [source]

```
instance : Module ℂ UpSinglet
```

def valLinEquiv [source]

The linear equivalence between `UpSinglet` and its underlying tensor product space.

```
def valLinEquiv : UpSinglet ≃ₗ[ℂ]
    Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3) where
```

lemma valLinEquiv_apply [source]

```
@[simp]
lemma valLinEquiv_apply (q : UpSinglet) : valLinEquiv q = q.val
```

lemma valLinEquiv_symm_apply [source]

```
lemma valLinEquiv_symm_apply (m : Fermion.RightHandedWeyl ⊗[ℂ] EuclideanSpace ℂ (Fin 3)) :
    valLinEquiv.symm m = ⟨m⟩
```

lemma val_add [source]

```
@[simp]
lemma val_add (q1 q2 : UpSinglet) : (q1 + q2).val = q1.val + q2.val
```

lemma val_smul [source]

```
@[simp]
lemma val_smul (r : ℂ) (q : UpSinglet) : (r • q).val = r • q.val
```

def repLorentzGroup [source]

The representation of the Lorentz group on the space of up-type quark fields.

```
noncomputable def repLorentzGroup : Representation ℂ (SL(2,ℂ)) UpSinglet where
```

def repGaugeGroupI [source]

The action of the full Standard Model gauge group on up-type quark fields.

```
noncomputable def repGaugeGroupI : Representation ℂ GaugeGroupI UpSinglet where
```

lemma repGaugeGroupI_tmul [source]

```
lemma repGaugeGroupI_tmul (g : GaugeGroupI) (ψ : Fermion.RightHandedWeyl)
    (v : EuclideanSpace ℂ (Fin 3)) :
    repGaugeGroupI g ⟨ψ ⊗ₜ v⟩ = ⟨g.toU1 ^ 4 • ψ ⊗ₜ (g.toSU3.1.toEuclideanLin v)⟩
```

lemma repGaugeGroupI_tmul_basis_eq_sum [source]

The action of the full gauge group on a tensor product of basis elements, expanded as a sum over the columns of the `SU(3)` matrix.

```
lemma repGaugeGroupI_tmul_basis_eq_sum (g : GaugeGroupI) (k : Fin 2) (i : Fin 3) :
    repGaugeGroupI g ⟨RightHandedWeyl.basis k ⊗ₜ[ℂ] EuclideanSpace.basisFun (Fin 3) ℂ i⟩ =
      ∑ i' : Fin 3, (g.toU1.1 ^ 4  * g.toSU3.1 i' i)
      • ⟨(RightHandedWeyl.basis k ⊗ₜ[ℂ] EuclideanSpace.basisFun (Fin 3) ℂ i') : UpSinglet⟩
```

lemma repGaugeGroupI_eq_iff_mul_eq [source]

```
lemma repGaugeGroupI_eq_iff_mul_eq {g1 g2 : GaugeGroupI} :
    repGaugeGroupI g1 = repGaugeGroupI g2 ↔ ∀ i i',
    g1.toU1.1 ^ 4 * g1.toSU3.1 i' i = g2.toU1.1 ^ 4 * g2.toSU3.1 i' i
```

lemma mem_repGaugeGroupI_ker_iff_eq [source]

```
lemma mem_repGaugeGroupI_ker_iff_eq {g : GaugeGroupI} :
    g ∈ repGaugeGroupI.ker ↔ ∃ a : ℂ, g.toSU3.1 = a • 1 ∧  a * g.toU1.1 ^ 4 = 1
```

lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI [source]

```
lemma gaugeGroup_subgroup_ℤ₆_le_ker_repGaugeGroupI :
    GaugeGroupQuot.subgroup .ℤ₆ ≤ repGaugeGroupI.ker
```

lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI [source]

```
lemma gaugeGroup_subgroup_le_ker_repGaugeGroupI (Q : GaugeGroupQuot) :
    Q.subgroup ≤ repGaugeGroupI.ker
```

def repGaugeGroup [source]

The action of the Standard Model gauge group, potentially quotiented by a discrete factor on quark fields.

```
noncomputable def repGaugeGroup : (Q : GaugeGroupQuot) →
    Representation ℂ (GaugeGroup Q) UpSinglet
  | .I => repGaugeGroupI
  | .ℤ₆ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₆)
  | .ℤ₂ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₂)
  | .ℤ₃ => QuotientGroup.lift _ repGaugeGroupI (gaugeGroup_subgroup_le_ker_repGaugeGroupI .ℤ₃)
```

2.44 `Physlib.QuantumMechanics.FreeParticle.Basic`

Physlib/QuantumMechanics/FreeParticle/Basic.lean on GitHub.

`structure FreeParticle` [source]

A free, spinless quantum particle with mass `m > 0` in `Space d`.

```
structure FreeParticle (d : ℕ) where
  m : ℝ
  hm : 0 < m
```

`lemma m_pos` [source]

```
@[simp]
lemma m_pos : 0 < Q.m
```

`lemma m_nonneg` [source]

```
@[simp]
lemma m_nonneg : 0 ≤ Q.m
```

`lemma m_ne_zero` [source]

```
@[simp]
lemma m_ne_zero : Q.m ≠ 0
```

`abbrev HS` [source]

The Hilbert space for the free particle.

```
@[nolint unusedArguments]
abbrev HS (_ : FreeParticle d) : Type _
```

`def hamiltonian` [source]

The Hamiltonian, `p²/(2m)`.

```
def hamiltonian : Q.HS →ₗ.[ℂ] Q.HS
```

`informal_lemma hamiltonian_essentially_self_adjoint` [source]

The Hamiltonian for the free particle is essentially self-adjoint. This follows immediately from the ess. self-adjointness of the momentum-square operator.

```
informal_lemma hamiltonian_essentially_self_adjoint where
  deps := [``FreeParticle]
  tag := "QM-FP-hamESA"
```

`informal_definition toQuantumSystem` [source]

The free particle as a quantum system (self-adjoint Hamiltonian acting on a Hilbert space).

```
informal_definition toQuantumSystem where
  deps := [``FreeParticle]
  tag := "QM-FP-sys"
```

2.45 Physlib.QuantumMechanics.HarmonicOscillator.Basic

Physlib/QuantumMechanics/HarmonicOscillator/Basic.lean on GitHub.

structure HarmonicOscillator [source]

The d-dimensional quantum harmonic oscillator.

```
structure HarmonicOscillator (d : ℕ) where
  m : ℝ
  hm : 0 < m
  ω : Fin d → ℝ
  hω : ∀ i, 0 < ω i
```

lemma m_pos [source]

```
@[simp]
lemma m_pos : 0 < Q.m
```

lemma m_nonneg [source]

```
@[simp]
lemma m_nonneg : 0 ≤ Q.m
```

lemma m_ne_zero [source]

```
@[simp]
lemma m_ne_zero : Q.m ≠ 0
```

lemma ω_pos [source]

```
@[simp]
lemma ω_pos : 0 < Q.ω i
```

lemma ω_nonneg [source]

```
@[simp]
lemma ω_nonneg : 0 ≤ Q.ω i
```

lemma ω_ne_zero [source]

```
@[simp]
lemma ω_ne_zero : Q.ω i ≠ 0
```

def ξ [source]

The characteristic length `ξ i = √ħ / (√Q.m * √(Q.ω i))`.

```
def ξ : ℝ
```

lemma ξ_eq [source]

```
lemma ξ_eq : Q.ξ i = √ħ / (√Q.m * √(Q.ω i))
```

lemma ξ_pos [source]

```
@[simp]
lemma ξ_pos : 0 < Q.ξ i
```

lemma ξ_nonneg [source]

```
@[simp]
lemma ξ_nonneg : 0 ≤ Q.ξ i
```

lemma ξ_ne_zero [source]

```
@[simp]
lemma ξ_ne_zero : Q.ξ i ≠ 0
```

lemma ξ_sq [source]

```
lemma ξ_sq : (Q.ξ i) ^ 2 = ħ / (Q.m * Q.ω i)
```

lemma ξ_inv [source]

```
lemma ξ_inv : (Q.ξ i)⁻¹ = √Q.m * √(Q.ω i) / √ħ
```

lemma ξ_inv' [source]

```
lemma ξ_inv' : (Q.ξ i)⁻¹ = Q.m * Q.ω i * Q.ξ i / ħ
```

def potentialMatrix [source]

The positive-definite matrix defining the quadratic potential function.

```
def potentialMatrix : Matrix (Fin d) (Fin d) ℝ
```

lemma potentialMatrix_eq [source]

```
lemma potentialMatrix_eq : Q.potentialMatrix = diagonal ((2⁻¹ * Q.m) • Q.ω ^ 2)
```

lemma potentialMatrix_isHermitian [source]

```
lemma potentialMatrix_isHermitian : Q.potentialMatrix.IsHermitian
```

lemma potentialMatrix_mulVec [source]

```
@[simp]
lemma potentialMatrix_mulVec (v : Fin d → ℝ) :
    Q.potentialMatrix *ᵥ v = (2⁻¹ * Q.m) • (Q.ω ^ 2 * v)
```

def potentialQuadraticForm [source]

The positive-definite quadratic form associated to the potential matrix.

```
def potentialQuadraticForm : QuadraticForm ℝ (Fin d → ℝ)
```

def potentialFunction [source]

The quadratic potential function, $\frac{1}{2}m \cdot \sum i, \omega_i^2 \cdot x_i^2$.

```
def potentialFunction : Space d → ℝ
```

lemma potentialFunction_eq [source]

```
lemma potentialFunction_eq : Q.potentialFunction = Q.potentialQuadraticForm ∘ Space.val
```

informal_lemma potentialFunction_aestronglyMeasurable [source]

The potential function for the harmonic oscillator is a.e. strongly measurable.

```
informal_lemma potentialFunction_aestronglyMeasurable where
  deps := [``HarmonicOscillator]
  tag := "QM-HO-potAESM"
```

def IsIsotropic [source]

A Harmonic oscillator is isotropic if all natural frequencies are equal.

```
def IsIsotropic : Prop
```

lemma isIsotropic_def [source]

```
lemma isIsotropic_def : Q.IsIsotropic ↔ ∀ i j, Q.ω i = Q.ω j
```

lemma isIsotropic_of_one [source]

```
lemma isIsotropic_of_one (Q : HarmonicOscillator 1) : Q.IsIsotropic
```

abbrev HS [source]

The Hilbert space for the quantum harmonic oscillator.

```
@[nolint unusedArguments]
abbrev HS (_ : HarmonicOscillator d) : Type _
```

def kineticOperator [source]

The kinetic energy operator, `p²/2m`.

```
def kineticOperator : Q.HS →ₗ.[ℂ] Q.HS
```

def potentialOperator [source]

The potential operator which maps `ψ` to `Q.potentialFunction • ψ`.

```
def potentialOperator : Q.HS →ₗ.[ℂ] Q.HS
```

informal_lemma potentialOperator_isSelfAdjoint [source]

The potential operators for the harmonic oscillator is self-adjoint.

```
informal_lemma potentialOperator_isSelfAdjoint where
  deps := [``HarmonicOscillator.potentialFunction_aestronglyMeasurable]
  tag := "QM-HO-potSA"
```

def hamiltonian [source]

The Hamiltonian for the harmonic oscillator.

```
def hamiltonian : Q.HS →ₗ.[ℂ] Q.HS
```

lemma hamiltonain_eq [source]

```
lemma hamiltonain_eq : Q.hamiltonian = Q.kineticOperator + Q.potentialOperator
```

informal_lemma hamiltonian_essentially_self_adjoint [source]

The Hamiltonian for the harmonic oscillator is essentially self-adjoint.

```
informal_lemma hamiltonian_essentially_self_adjoint where
  deps := [``HarmonicOscillator.hamiltonian]
  tag := "QM-HO-hamESA"
```

informal_definition toQuantumSystem [source]

The `d`-dimensional harmonic oscillator as a quantum system (self-adjoint Hamiltonian acting on a Hilbert space).

```
informal_definition toQuantumSystem where
  deps := [``HarmonicOscillator.hamiltonian_essentially_self_adjoint]
  tag := "QM-HO-sys"
```

2.46 Physlib.QuantumMechanics.HarmonicOscillator.OneDimension.Basic

Physlib/QuantumMechanics/HarmonicOscillator/OneDimension/Basic.lean on GitHub.

lemma m_pos [source]

```
@[simp]
lemma m_pos : 0 < Q.m
```

lemma m_nonneg [source]

```
@[simp]
lemma m_nonneg : 0 ≤ Q.m
```

lemma ω_pos [source]

```
@[simp]
lemma ω_pos : 0 < Q.ω
```

lemma ω_nonneg [source]

```
@[simp]
lemma ω_nonneg : 0 ≤ Q.ω
```

lemma ω_ne_zero [source]

```
@[simp]
lemma ω_ne_zero : Q.ω ≠ 0
```

2.47 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.Basic

Physlib/QuantumMechanics/HilbertSpaces/SpaceD/Basic.lean on GitHub.

lemma toBra_apply_apply [source]

```
@[simp]
lemma toBra_apply_apply : toBra ψ φ = ⟪ψ, φ⟫_ℂ
```

lemma toBra_symm_apply [source]

```
@[simp]
lemma toBra_symm_apply (f : StrongDual ℂ (SpaceDHilbertSpace d μ)) : ⟪toBra.symm f, ψ⟫_ℂ = f ψ
```

lemma memHS_coe [source]

Elements of the Hilbert space satisfy the property `MemHS`.

```
lemma memHS_coe : MemHS ψ μ
```

lemma zero [source]

```
@[simp]
lemma MemHS.zero : MemHS (0 : Space d → ℂ) μ
```

lemma neg [source]

```
lemma MemHS.neg (hf : MemHS f μ) : MemHS (-f) μ
```

lemma memHS_neg_iff [source]

```
lemma memHS_neg_iff : MemHS (-f) μ ↔ MemHS f μ
```

lemma add [source]

```
lemma MemHS.add (hf : MemHS f μ) (hg : MemHS g μ) : MemHS (f + g) μ
```

lemma sub [source]

```
lemma MemHS.sub (hf : MemHS f μ) (hg : MemHS g μ) : MemHS (f - g) μ
```

lemma const_smul [source]

```
lemma MemHS.const_smul (c : ℂ) (hf : MemHS f μ) : MemHS (c • f) μ
```

lemma memHS_const_smul_iff [source]

```
lemma memHS_const_smul_iff {c : ℂ} (hc : c ≠ 0) : MemHS (c • f) μ ↔ MemHS f μ
```

lemma ae_eq [source]

```
lemma MemHS.ae_eq (hfg : f =ᵐ[μ] g) (hf : MemHS f μ) : MemHS g μ
```

lemma memHS_congr_ae [source]

```
lemma memHS_congr_ae (hfg : f =ᵐ[μ] g) : MemHS f μ ↔ MemHS g μ
```

lemma congr_norm [source]

```
lemma MemHS.congr_norm
    (hf : MemHS f μ) (hg : AEStronglyMeasurable g μ) (hfg : ∀ᵐ x ∂μ, ‖f x‖ = ‖g x‖) : MemHS g μ
```

lemma memHS_congr_norm [source]

```
lemma memHS_congr_norm (hf : AEStronglyMeasurable f μ) (hg : AEStronglyMeasurable g μ)
    (hfg : ∀ᵐ x ∂μ, ‖f x‖ = ‖g x‖) : MemHS f μ ↔ MemHS g μ
```

lemma mono [source]

```
lemma MemHS.mono (hf : MemHS f μ) (hg : AEStronglyMeasurable g μ) (hfg : ∀ᵐ x ∂μ, ‖g x‖ ≤ ‖f x
    ‖) :
    MemHS g μ
```

lemma mono_measure [source]

```
lemma MemHS.mono_measure (h : μ' ≤ μ) (hf : MemHS f μ) : MemHS f μ'
```

lemma restrict [source]

```
lemma MemHS.restrict (Ω : Set (Space d)) (hf : MemHS f μ) : MemHS f (μ.restrict Ω)
```

lemma mono_restrict [source]

```
lemma MemHS.mono_restrict {Ω Ω' : Set (Space d)} (h : Ω' ≤ Ω) (hf : MemHS f (μ.restrict Ω)) :
    MemHS f (μ.restrict Ω')
```

lemma indicator [source]

```
lemma MemHS.indicator {Ω : Set (Space d)} (hΩ : MeasurableSet Ω) (hf : MemHS f μ) :
    MemHS (Ω.indicator f) μ
```

lemma indicator_of_restrict [source]

If `f` is a member of the Hilbert space with measure restricted to `Ω` then the representative `Ω.indicator f` which vanishes outside `Ω` is a member of `SpaceDHilbertSpace d μ`.

```
lemma MemHS.indicator_of_restrict
    {Ω : Set (Space d)} (hΩ : MeasurableSet Ω) (hf : MemHS f (μ.restrict Ω)) :
    MemHS (Ω.indicator f) μ
```

lemma mk_neg [source]

```
@[simp]
lemma mk_neg : mk hf.neg = -mk hf
```

lemma mk_sub [source]

```
@[simp]
lemma mk_sub : mk (hf.sub hg) = mk hf - mk hg
```

lemma coeFn_mk [source]

```
lemma coeFn_mk : mk hf =ᵐ[μ] f
```

lemma coeFn_zero [source]

```
lemma coeFn_zero : ⇑(0 : SpaceDHilbertSpace d μ) =ᵐ[μ] 0
```

lemma coeFn_neg [source]

```
lemma coeFn_neg : ⇑(-ψ) =ᵐ[μ] -ψ
```

lemma coeFn_add [source]

```
lemma coeFn_add : ↑(ψ.val + φ.val) =ᵐ[μ] ψ + φ
```

lemma coeFn_sub [source]

```
lemma coeFn_sub : ↑(ψ.val - φ.val) =ᵐ[μ] ψ - φ
```

lemma coeFn_smul [source]

```
lemma coeFn_smul : ↑(c • ψ) =ᵐ[μ] c • ψ
```

abbrev toTemperedDistribution [source]

The tempered distribution associated to a state.

```
abbrev toTemperedDistribution [μ.HasTemperateGrowth]
    (ψ : SpaceDHilbertSpace d μ) : 𝒮'(Space d, ℂ)
```

abbrev toTemperedDistributionCLM [source]

The embedding of states into tempered distributions as a continuous linear map.

```
abbrev toTemperedDistributionCLM (d : ℕ) (μ : Measure (Space d) := volume)
    [μ.HasTemperateGrowth] : SpaceDHilbertSpace d μ →L[ℂ] 𝒮'(Space d, ℂ)
```

abbrev SpaceDHilbertSpaceOn [source]

The L^2-space `SpaceDHilbertSpace d (μ.restrict Ω)`.

Elements are equivalence classes of functions which agree `μ`-a.e. on `Ω` and which have `∫ x in Ω, ‖f x‖ ^ 2 ∂μ` finite.

```
abbrev SpaceDHilbertSpaceOn {d : ℕ} (Ω : Set (Space d)) (μ : Measure (Space d) := volume)
```

def subspaceProjection [source]

The linear map projecting `SpaceDHilbertSpace d μ` onto `SpaceDHilbertSpaceOn Ω μ`.

```
def subspaceProjection : SpaceDHilbertSpace d μ →ₗ[ℂ] SpaceDHilbertSpaceOn Ω μ where
```

lemma subspaceProjection_apply [source]

```
lemma subspaceProjection_apply : subspaceProjection Ω μ ψ =ᵐ[μ.restrict Ω] ψ
```

lemma subspaceProjection_norm_le [source]

```
lemma subspaceProjection_norm_le : ‖subspaceProjection Ω μ ψ‖ ≤ ‖ψ‖
```

def subspaceIncl [source]

The linear isometry including `SpaceDHilbertSpaceOn Ω μ` as a sub-Hilbert space of `SpaceDHilbertSpace d μ`, defined by mapping `ψ` to `Ω.indicator ψ`.

```
def subspaceIncl : SpaceDHilbertSpaceOn Ω μ →ₗᵢ[ℂ] SpaceDHilbertSpace d μ where
```

lemma subspaceIncl_apply [source]

```
lemma subspaceIncl_apply : subspaceIncl hΩ μ φ =ᵐ[μ] Ω.indicator φ
```

lemma leftInverse_subspaceProjection [source]

```
lemma leftInverse_subspaceProjection :
    LeftInverse (subspaceProjection Ω μ) (subspaceIncl hΩ μ)
```

lemma subspaceProjection_subspaceIncl_apply [source]

```
@[simp]
lemma subspaceProjection_subspaceIncl_apply : subspaceProjection Ω μ (subspaceIncl hΩ μ φ) = φ
```

lemma subspaceProjection_surjective [source]

```
lemma subspaceProjection_surjective : Surjective (subspaceProjection Ω μ)
```

2.48 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.DirichletSubmodule

Physlib/QuantumMechanics/HilbertSpaces/SpaceD/DirichletSubmodule.lean on GitHub.

def DirichletSchwartzMap [source]

The Schwartz maps which vanish on `frontier Ω`.

```
def DirichletSchwartzMap : Submodule ℂ 𝓢(Space d, ℂ) where
```

abbrev DirichletSubmoduleOn [source]

The submodule of the Hilbert space on `Ω` consisting of the equivalence classes of Schwartz maps which vanish on `frontier Ω`.

```
abbrev DirichletSubmoduleOn : Submodule ℂ (SpaceDHilbertSpaceOn Ω μ)
```

lemma mem_iff [source]

```
lemma mem_iff {ψ : SpaceDHilbertSpaceOn Ω μ} :
    ψ ∈ DirichletSubmoduleOn Ω μ ↔
      ∃ f : DirichletSchwartzMap Ω, subspaceProjection Ω μ (schwartzIncl μ f) = ψ
```

lemma le_schwartzSubmoduleOn [source]

```
lemma le_schwartzSubmoduleOn : DirichletSubmoduleOn Ω μ ≤ SchwartzSubmoduleOn Ω μ
```

2.49 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.Fourier

Physlib/QuantumMechanics/HilbertSpaces/SpaceD/Fourier.lean on GitHub.

def fourierUnitary [source]

The L^2 Fourier transform as a unitary on `SpaceDHilbertSpace d`.

```
noncomputable def fourierUnitary (d : ℕ) :
    SpaceDHilbertSpace d ≃ₗᵢ[ℂ] SpaceDHilbertSpace d
```

lemma fourierUnitary_apply [source]

`fourierUnitary d` acts as the L^2 Fourier transform $\mathcal{F}$.

```
@[simp]
lemma fourierUnitary_apply (ψ : SpaceDHilbertSpace d) : fourierUnitary d ψ = 𝓕 ψ
```

lemma fourierUnitary_symm_apply [source]

`(fourierUnitary d).symm` acts as the inverse L^2 Fourier transform $\mathcal{F}^{-}$.

```
@[simp]
lemma fourierUnitary_symm_apply (ψ : SpaceDHilbertSpace d) : (fourierUnitary d).symm ψ = 𝓕⁻ ψ
```

lemma schwartzIncl_fourier_eq [source]

Applying `fourierUnitary d` to the L^2 class of a Schwartz map `f` gives the L^2 class of the Schwartz Fourier transform $\mathcal{F}$ `f`.

```
lemma schwartzIncl_fourier_eq (f : 𝓢(Space d, ℂ)) :
    𝓕 (schwartzIncl volume f) = schwartzIncl volume (𝓕 f)
```

lemma schwartzIncl_fourierInv_eq [source]

Applying $\mathcal{F}^{-}$ to the L^2 class of a Schwartz map `f` gives the L^2 class of the inverse Schwartz Fourier transform $\mathcal{F}^{-}$ `f`.

```
lemma schwartzIncl_fourierInv_eq (f : 𝓢(Space d, ℂ)) :
    𝓕⁻ (schwartzIncl volume f) = schwartzIncl volume (𝓕⁻ f)
```

lemma fourierInv_schwartzIncl_fourier [source]

Pulling the L^2 class of $\mathcal{F}$ `f` back through the Fourier unitary recovers the L^2 class of `f`.

```
@[simp]
lemma fourierInv_schwartzIncl_fourier (f : 𝓢(Space d, ℂ)) :
    𝓕⁻ (schwartzIncl volume (𝓕 f)) = schwartzIncl volume f
```

lemma fourierUnitary_map_schwartzSubmodule [source]

The Fourier unitary maps the Schwartz submodule onto itself.

```
lemma fourierUnitary_map_schwartzSubmodule :
    (SchwartzSubmodule d).map (fourierUnitary d).toLinearMap = SchwartzSubmodule d
```

2.50 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.PolyBddSchwartzSubmodule

Physlib/QuantumMechanics/HilbertSpaces/SpaceD/PolyBddSchwartzSubmodule.lean on GitHub.

def PolyBddSchwartzMap [source]

A function is a bounded Schwartz map if it is both Schwartz and bounded by powers of $\|x\|$.

```
def PolyBddSchwartzMap (d : ℕ) (a : ℕ∞) : Submodule ℂ 𝓢(Space d, ℂ) where
```

abbrev PolyBddSchwartzSubmodule [source]

The submodule of `SpaceDHilbertSpace d` corresponding to bounded Schwartz maps.

```
abbrev PolyBddSchwartzSubmodule
    (d : ℕ) (a : ℕ∞) (μ : Measure (Space d) := volume) [μ.HasTemperateGrowth] :
    Submodule ℂ (SpaceDHilbertSpace d μ)
```

instance CoeOut (PolyBddSchwartzMap d a) 𝓢(Space d, ℂ) [source]

```
instance : CoeOut (PolyBddSchwartzMap d a) 𝓢(Space d, ℂ)
```

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

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

lemma PolyBddSchwartzMap_zero_eq_top [source]

```
lemma PolyBddSchwartzMap_zero_eq_top (d : ℕ) : PolyBddSchwartzMap d 0 = ⊤
```

lemma PolyBddSchwartzMap_antitone [source]

```
lemma PolyBddSchwartzMap_antitone (d : ℕ) {a b : ℕ∞} (h : a ≤ b) :
    PolyBddSchwartzMap d b ≤ PolyBddSchwartzMap d a
```

lemma le_SchwartzSubmodule [source]

```
lemma le_SchwartzSubmodule (d : ℕ) (a : ℕ∞) : PolyBddSchwartzSubmodule d a ≤ SchwartzSubmodule
    d
```

2.51 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.SchwartzSubmodule

Physlib/QuantumMechanics/HilbertSpaces/SpaceD/SchwartzSubmodule.lean on GitHub.

abbrev SchwartzSubmodule [source]

The submodule of `SpaceDHilbertSpace d` corresponding to Schwartz maps.

```
abbrev SchwartzSubmodule (d : ℕ) (μ : Measure (Space d) := volume) [μ.HasTemperateGrowth] :
    Submodule ℂ (SpaceDHilbertSpace d μ)
```

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

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

lemma toTemperedDistribution_schwartzIncl_eq [source]

```
lemma toTemperedDistribution_schwartzIncl_eq :
    toTemperedDistribution (schwartzIncl μ g) = g.toTemperedDistributionCLM (Space d) ℂ μ
```

abbrev SchwartzSubmoduleOn [source]

The projection of `SchwartzSubmodule d μ` onto `SpaceDHilbertSpaceOn Ω μ`.

```
abbrev SchwartzSubmoduleOn
    {d : ℕ} (Ω : Set (Space d)) (μ : Measure (Space d) := volume) [μ.HasTemperateGrowth] :
    Submodule ℂ (SpaceDHilbertSpaceOn Ω μ)
```

lemma mem_iff [source]

```
lemma mem_iff {ψ : SpaceDHilbertSpaceOn Ω μ} :
    ψ ∈ SchwartzSubmoduleOn Ω μ ↔ ∃ φ : SchwartzSubmodule d μ, subspaceProjection Ω μ φ = ψ
```

2.52 Physlib.QuantumMechanics.HilbertSpaces.SpaceD.SobolevSubmodule

Physlib/QuantumMechanics/HilbertSpaces/SpaceD/SobolevSubmodule.lean on GitHub.

def SobolevSubmodule [source]

The Sobolev space `H^s` as a submodule of `SpaceDHilbertSpace d`: the L^2 classes whose associated tempered distribution satisfies `MemSobolev s 2`.

```
def SobolevSubmodule (d : ℕ) (s : ℝ) : Submodule ℂ (SpaceDHilbertSpace d) where
```

lemma mem_sobolevSubmodule_iff [source]

Membership in `H^s` is the Sobolev condition on the associated tempered distribution.

```
lemma mem_sobolevSubmodule_iff {s : ℝ} {ψ : SpaceDHilbertSpace d} :
    ψ ∈ SobolevSubmodule d s ↔ MemSobolev s 2 (toTemperedDistribution ψ)
```

lemma schwartzIncl_mem [source]

Schwartz maps lie in every Sobolev space `H^s`.

```
lemma SobolevSubmodule.schwartzIncl_mem (s : ℝ) (g : 𝓢(Space d, ℂ)) :
    schwartzIncl volume g ∈ SobolevSubmodule d s
```

lemma schwartzSubmodule_le_sobolevSubmodule [source]

The Schwartz submodule is contained in every Sobolev space `H^s`.

```
lemma schwartzSubmodule_le_sobolevSubmodule (s : ℝ) :
    SchwartzSubmodule d ≤ SobolevSubmodule d s
```

lemma dense [source]

Every Sobolev space `H^s` is dense in `SpaceDHilbertSpace d`, containing the dense Schwartz submodule.

```
lemma SobolevSubmodule.dense (s : ℝ) :
    Dense (SobolevSubmodule d s : Set (SpaceDHilbertSpace d))
```

lemma antitone [source]

The Sobolev spaces shrink as the regularity index grows: `H^s ≤ H^s'` for `s' ≤ s`.

```
lemma SobolevSubmodule.antitone (d : ℕ) : Antitone (SobolevSubmodule d)
```

2.53 Physlib.QuantumMechanics.InfiniteSquareWell.Basic

Physlib/QuantumMechanics/InfiniteSquareWell/Basic.lean on GitHub.

structure InfiniteSquareWell [source]

A spinless quantum particle with mass `m > 0` confined to a cuboid in `Space d`.

The bounds of the well are specified by two functions `lower upper : Fin d → ℝ` satisfying `∀ i, lower i < upper i`.

```
structure InfiniteSquareWell (d : ℕ) where
  m : ℝ
  hm : 0 < m
  lower : Fin d → ℝ
  upper : Fin d → ℝ
  h_bounds : ∀ i, lower i < upper i
```

lemma m_pos [source]

```
@[simp]
lemma m_pos : 0 < Q.m
```

lemma m_nonneg [source]

```
@[simp]
lemma m_nonneg : 0 ≤ Q.m
```

lemma m_ne_zero [source]

```
@[simp]
lemma m_ne_zero : Q.m ≠ 0
```

def well [source]

The domain of the infinite square well as a Cartesian product of closed intervals.

```
def well : Set (Space d)
```

def measure [source]

The measure associated with the domain of the infinite square well.

```
def measure : Measure (Space d)
```

abbrev HS [source]

The Hilbert space for the infinite square well.

```
abbrev HS : Type _
```

informal_definition hamiltonian [source]

The Hamiltonian for the infinite square well is `(2m)⁻¹momentumSqOperator` with respect to `InfiniteSquareWell.measure`. This requires first generalizing `momentumSqOperator` to `Space d` measures other than `volume`.

```
informal_definition hamiltonian where
  deps := [``InfiniteSquareWell]
  tag := "QM-ISW-ham"
```

informal_lemma hamiltonian_essentially_self_adjoint [source]

The Hamiltonian for the infinite square well is essentially self-adjoint.

```
informal_lemma hamiltonian_essentially_self_adjoint where
  deps := [``InfiniteSquareWell]
  tag := "QM-ISW-hamESA"
```

informal_definition toQuantumSystem [source]

The particle in an infinite square well as a quantum system (self-adjoint Hamiltonian acting on a Hilbert space).

```
informal_definition toQuantumSystem where
  deps := [``InfiniteSquareWell]
  tag := "QM-ISW-sys"
```

2.54 Physlib.QuantumMechanics.Operators.Multiplication

Physlib/QuantumMechanics/Operators/Multiplication.lean on GitHub.

lemma mulOperator_domain_eq_top [source]

The multiplication operator of a μ-a.e. bounded function has full domain.

```
lemma mulOperator_domain_eq_top {μ : Measure (Space d)}
    {f : Space d → ℂ} (hf : AEStronglyMeasurable f μ) {c : ℝ} (hfc : ∀ᵐ x ∂μ, ‖f x‖ ≤ c) :
    (𝓜 μ f).domain = ⊤
```

lemma mulOperator_domain_antitone [source]

The domains of multiplication operators shrink with increasing function norm.

```
lemma mulOperator_domain_antitone {μ : Measure (Space d)}
    {f g : Space d → ℂ} (hg : AEStronglyMeasurable g μ) (h : ∀ᵐ x ∂μ, ‖g x‖ ≤ ‖f x‖) :
    (𝓜 μ f).domain ≤ (𝓜 μ g).domain
```

lemma mulOperator_domain_eq_of_congr_norm [source]

The multiplication operators corresponding to functions of μ-a.e. equal norm have the same domain.

```
lemma mulOperator_domain_eq_of_congr_norm {μ : Measure (Space d)} {f g : Space d → ℂ}
    (hf : AEStronglyMeasurable f μ) (hg : AEStronglyMeasurable g μ) (h : ∀ᵐ x ∂μ, ‖f x‖ = ‖g x
    ‖) :
    (𝓜 μ f).domain = (𝓜 μ g).domain
```

lemma mulOperator_isClosed [source]

Multiplication operators of μ-a.e. strongly measurable functions are closed.

```
lemma mulOperator_isClosed {μ : Measure (Space d)} [IsFiniteMeasureOnCompacts μ]
    {f : Space d → ℂ} (hf : AEStronglyMeasurable f μ) :
    (𝓜 μ f).IsClosed
```

lemma mulOperator_zero [source]

The multiplication operator of the zero function is the zero operator (domain ⊤).

```
@[simp]
lemma mulOperator_zero (μ : Measure (Space d)) : 𝓜 μ 0 = 0
```

lemma mulOperator_eq_of_congr_ae [source]

μ-a.e. equal functions give rise to the same multiplication operator.

```
lemma mulOperator_eq_of_congr_ae {μ : Measure (Space d)} {f g : Space d → ℂ} (h : f =ᵐ[μ] g) :
    𝓜 μ f = 𝓜 μ g
```

lemma mulOperator_const_smul_ge [source]

Scalar multiplication and `mulOperator` commute except possibly for `c = 0` where the domains of `0 • 𝓜 μ f` and `𝓜 μ 0 = 0` may not agree.

See `mulOperator_const_smul_eq` for equality when `c ≠ 0`.

```
lemma mulOperator_const_smul_ge (μ : Measure (Space d)) (f : Space d → ℂ) (c : ℂ) :
    c • 𝓜 μ f ≤ 𝓜 μ (c • f)
```

lemma mulOperator_const_smul_eq [source]

Scalar multiplication and `mulOperator` commute for `c ≠ 0`.

```
@[simp]
lemma mulOperator_const_smul_eq (μ : Measure (Space d)) (f : Space d → ℂ) {c : ℂ} (hc : c ≠ 0)
    :
    𝓜 μ (c • f) = c • 𝓜 μ f
```

lemma mulOperator_neg [source]

Negation and `mulOperator` commute.

```
@[simp]
lemma mulOperator_neg (μ : Measure (Space d)) (f : Space d → ℂ) : 𝓜 μ (-f) = -𝓜 μ f
```

lemma mulOperator_add_ge [source]

`𝓜 μ (f + g)` extends `𝓜 μ f + 𝓜 μ g`.

In general the domains do not match: `ψ ∈ (𝓜 μ f + 𝓜 μ g).domain` amounts to `MemHS (f • ψ) μ` _and_ `MemHS (g • ψ) μ` whereas `ψ ∈ (𝓜 μ (f + g)).domain` is equivalent to the weaker condition `MemHS ((f + g) • ψ) μ`.

See `mulOperator_add_eq` and `mulOperator_add_eq'` for sufficient conditions to ensure equality.

```
lemma mulOperator_add_ge (μ : Measure (Space d)) (f g : Space d → ℂ) :
    𝓜 μ f + 𝓜 μ g ≤ 𝓜 μ (f + g)
```

lemma mulOperator_add_eq [source]

`(𝓜 μ g).domain = ⊤` is a sufficient condition to ensure equality in `mulOperator_add_ge`.

```
@[simp]
lemma mulOperator_add_eq
    {μ : Measure (Space d)} (f : Space d → ℂ) {g : Space d → ℂ} (h : (𝓜 μ g).domain = ⊤) :
    𝓜 μ (f + g) = 𝓜 μ f + 𝓜 μ g
```

lemma mulOperator_add_eq' [source]

Having both `‖f x‖ ≤ c * ‖f x + g x‖` and `‖g x‖ ≤ c' * ‖f x + g x‖` μ-a.e. is a sufficient condition to ensure equality in `mulOperator_add_ge`.

```
lemma mulOperator_add_eq' {μ : Measure (Space d)}
    {f g : Space d → ℂ} (hf : AEStronglyMeasurable f μ) (hg : AEStronglyMeasurable g μ)
    (c c' : ℝ) (hf' : ∀ᵐ x ∂μ, ‖f x‖ ≤ c * ‖f x + g x‖) (hg' : ∀ᵐ x ∂μ, ‖g x‖ ≤ c' * ‖f x + g x
    ‖) :
    𝓜 μ (f + g) = 𝓜 μ f + 𝓜 μ g
```

lemma mulOperator_sub_ge [source]

`𝓜 μ (f - g)` extends `𝓜 μ f - 𝓜 μ g`.

In general the domains do not match: `ψ ∈ (𝓜 μ f - 𝓜 μ g).domain` amounts to `MemHS (f • ψ) μ` _and_ `MemHS (g • ψ) μ` whereas `ψ ∈ (𝓜 μ (f - g)).domain` is equivalent to the weaker condition `MemHS ((f - g) • ψ) μ`.

See `mulOperator_sub_eq` and `mulOperator_sub_eq'` for sufficient conditions to ensure equality.

```
lemma mulOperator_sub_ge (μ : Measure (Space d)) (f g : Space d → ℂ) :
    𝓜 μ f - 𝓜 μ g ≤ 𝓜 μ (f - g)
```

lemma mulOperator_sub_eq [source]

`(𝓜 μ g).domain = ⊤` is a sufficient condition to ensure equality in `mulOperator_sub_ge`.

```
@[simp]
lemma mulOperator_sub_eq
    {μ : Measure (Space d)} (f : Space d → ℂ) {g : Space d → ℂ} (h : (𝓜 μ g).domain = ⊤) :
    𝓜 μ (f - g) = 𝓜 μ f - 𝓜 μ g
```

lemma mulOperator_sub_eq' [source]

Having both `‖f x‖ ≤ c * ‖f x - g x‖` and `‖g x‖ ≤ c' * ‖f x - g x‖` μ-a.e. is a sufficient condition to ensure equality in `mulOperator_sub_ge`.

```
lemma mulOperator_sub_eq' {μ : Measure (Space d)}
    {f g : Space d → ℂ} (hf : AEStronglyMeasurable f μ) (hg : AEStronglyMeasurable g μ)
    (c c' : ℝ) (hf' : ∀ᵐ x ∂μ, ‖f x‖ ≤ c * ‖f x - g x‖) (hg' : ∀ᵐ x ∂μ, ‖g x‖ ≤ c' * ‖f x - g x
    ‖) :
    𝓜 μ (f - g) = 𝓜 μ f - 𝓜 μ g
```

lemma mulOperator_smul_ge [source]

`𝓜 μ (f • g)` extends `𝓜 μ f * 𝓜 μ g`.

In general the domains do not match: `ψ ∈ (𝓜 μ f * 𝓜 μ g).domain` amounts to `MemHS (g • ψ) μ` _and_ `MemHS (f • g • ψ) μ` whereas `ψ ∈ (𝓜 μ (f • g)).domain` only requires `MemHS (f • g • ψ) μ`.

See `mulOperator_smul_eq` for a sufficient condition to ensure equality.

```
lemma mulOperator_smul_ge (μ : Measure (Space d)) (f g : Space d → ℂ) :
    𝓜 μ f ∘ᵣ 𝓜 μ g ≤ 𝓜 μ (f • g)
```

lemma mulOperator_smul_eq [source]

`(𝓜 μ g).domain = ⊤` is a sufficient condition to ensure equality in `mulOperator_smul_ge`.

```
lemma mulOperator_smul_eq
    {μ : Measure (Space d)} (f : Space d → ℂ) {g : Space d → ℂ} (h : (𝓜 μ g).domain = ⊤) :
    𝓜 μ f ∘ᵣ 𝓜 μ g = 𝓜 μ (f • g)
```

2.55 Physlib.QuantumMechanics.Operators.OneDimension.Momentum

Physlib/QuantumMechanics/Operators/OneDimension/Momentum.lean on GitHub.

lemma momentumOperatorUnbounded_isSymmetric [source]

```
lemma momentumOperatorUnbounded_isSymmetric : momentumOperatorUnbounded.IsSymmetric
```

2.56 Physlib.QuantumMechanics.Operators.OneDimension.Parity

Physlib/QuantumMechanics/Operators/OneDimension/Parity.lean on GitHub.

lemma parityOperatorUnbounded_isSymmetric [source]

```
lemma parityOperatorUnbounded_isSymmetric :
    parityOperatorUnbounded.IsSymmetric
```

2.57 Physlib.QuantumMechanics.Operators.OneDimension.Position

Physlib/QuantumMechanics/Operators/OneDimension/Position.lean on GitHub.

lemma positionOperatorUnbounded_isSymmetric [source]

```
lemma positionOperatorUnbounded_isSymmetric :
    positionOperatorUnbounded.IsSymmetric
```

2.58 Physlib.QuantumMechanics.Operators.OneDimension.Unbounded

Physlib/QuantumMechanics/Operators/OneDimension/Unbounded.lean on GitHub.

def IsSymmetric [source]

*The condition for an unbounded operator to be *symmetric* (equivalently, formally self-adjoint): `⟪U ψ1, ι ψ2⟫ = ⟪ι ψ1, U ψ2⟫` for all `ψ1 ψ2`. This is the Hermitian pairing on the underlying space `S`; on its own it does **not** imply genuine self-adjointness (`A† = A`, including equality of domains), which for an unbounded operator on a proper dense core is strictly stronger.*

```
def IsSymmetric : Prop
```

2.59 Physlib.QuantumMechanics.Operators.SpectralTheory.SelfAdjoint

Physlib/QuantumMechanics/Operators/SpectralTheory/SelfAdjoint.lean on GitHub.

lemma mem_resolventSet_of_im_ne_zero [source]

Every non-real `z` lies in the resolvent set of a self-adjoint operator.

```
lemma mem_resolventSet_of_im_ne_zero {z : ℂ} (hz : z.im ≠ 0) : z ∈ ρ T
```

lemma sub_smul_surjective [source]

A self-adjoint operator has `T - z • 1` surjective for every non-real `z`: off the real axis a self-adjoint operator has `z` in its resolvent set, so `T - z • 1` has full range.

```
lemma sub_smul_surjective {z : ℂ} (hz : z.im ≠ 0) : Function.Surjective (T - z • 1).toFun
```

lemma unitaryConj_isSelfAdjoint [source]

Unitary conjugation preserves self-adjointness: if `A` is a self-adjoint operator on `H` and `u : H ≃ₗᵢ[ℂ] H'` is unitary, then `u A u⁻¹` is self-adjoint on `H'`. Symmetry, dense domain, and the two deficiency surjectivities of `A` all transfer through `u`.

```
lemma unitaryConj_isSelfAdjoint (u : H ≃ₗᵢ[ℂ] H') {A : H →ₗ.[ℂ] H} (hA : IsSelfAdjoint A) :
    IsSelfAdjoint (A.unitaryConj u)
```

2.60 Physlib.QuantumMechanics.Operators.SpectralTheory.Symmetric

Physlib/QuantumMechanics/Operators/SpectralTheory/Symmetric.lean on GitHub.

lemma isEssentiallySelfAdjoint_of_defectNumber_eq_zero [source]

The basic criterion for essential self-adjointness: a symmetric, densely-defined operator whose defect numbers at `I` and `-I` both vanish is essentially self-adjoint.

```
lemma isEssentiallySelfAdjoint_of_defectNumber_eq_zero
    (hdense : T.HasDenseDomain) (hpos : T.defectNumber I = 0) (hneg : T.defectNumber (-I) = 0)
    :
    T.IsEssentiallySelfAdjoint
```

2.61 Physlib.QuantumMechanics.Operators.Unbounded

Physlib/QuantumMechanics/Operators/Unbounded.lean on GitHub.

lemma isStarNormal_def [source]

```
lemma isStarNormal_def [CompleteSpace H] : IsStarNormal T ↔ T† * T = T * T†
```

lemma adjoint_domain_of_continuous [source]

```
lemma adjoint_domain_of_continuous [CompleteSpace H] (h : Continuous U) : U†.domain = ⊤
```

lemma adjoint_add_continuous [source]

```
lemma HasDenseDomain.adjoint_add_continuous [CompleteSpace H]
    (h₁ : T₁.HasDenseDomain) (h₂ : Continuous T₂) (h : T₁.domain ≤ T₂.domain) :
    (T₁ + T₂)† = T₁† + T₂†
```

lemma adjoint_sub_continuous [source]

```
lemma HasDenseDomain.adjoint_sub_continuous [CompleteSpace H]
    (h₁ : T₁.HasDenseDomain) (h₂ : Continuous T₂) (h : T₁.domain ≤ T₂.domain) :
    (T₁ - T₂)† = T₁† - T₂†
```

def LinearPMap.unitaryConj [source]

The conjugation `u A u⁻¹` of a partially-defined operator `A : H →ₗ.[ℂ] H` by a unitary `u : H ≃ₗᵢ[ℂ] H'`, with domain `u (A.domain) = u⁻¹ ⁻¹' (A.domain)` and action `y ↦ u (A (u⁻¹ y))`. Since `u` and `u⁻¹` are `ℂ`-linear, the result is again `ℂ`-linear.

```
def unitaryConj : H' →ₗ.[ℂ] H' where
```

lemma mem_unitaryConj_domain_iff [source]

Membership in the conjugated domain: `x ∈ D(u A u⁻¹) ↔ u⁻¹ x ∈ D(A)`.

```
lemma mem_unitaryConj_domain_iff {x : H'} :
    x ∈ (unitaryConj u A).domain ↔ u.symm x ∈ A.domain
```

lemma unitaryConj_apply [source]

The defining formula `(u A u⁻¹) x = u (A (u⁻¹ x))`.

```
lemma unitaryConj_apply (x : (unitaryConj u A).domain) :
    unitaryConj u A x = u (A ⟨u.symm (x : H'), (mem_unitaryConj_domain_iff u A).mp x.2⟩)
```

lemma map_mem_unitaryConj_domain [source]

`u` maps `D(A)` into `D(u A u⁻¹)`.

```
lemma map_mem_unitaryConj_domain (y : A.domain) : u (y : H) ∈ (unitaryConj u A).domain
```

lemma unitaryConj_apply_map [source]

The action on the image domain: `(u A u⁻¹)(u y) = u (A y)` for `y ∈ D(A)`.

```
lemma unitaryConj_apply_map (y : A.domain) :
    unitaryConj u A ⟨u (y : H), map_mem_unitaryConj_domain u A y⟩ = u (A y)
```

lemma IsFormalAdjoint.unitaryConj [source]

Unitary conjugation preserves formal adjointness. If `A` is a formal adjoint of `B`, then `u A u⁻¹` is a formal adjoint of `u B u⁻¹`. Unitary conjugation preserves symmetry when `A = B`.

```
lemma IsFormalAdjoint.unitaryConj {B : H →ₗ.[ℂ] H} (h : A.IsFormalAdjoint B) :
    (unitaryConj u A).IsFormalAdjoint (unitaryConj u B)
```

lemma unitaryConj_dense_domain [source]

If `A` has dense domain, then so does `u A u⁻¹`: the domain `u⁻¹ ⁻¹' (A.domain)` is the preimage of a dense set under a homeomorphism.

```
lemma HasDenseDomain.unitaryConj_dense_domain (hdense : A.HasDenseDomain) :
    (unitaryConj u A).HasDenseDomain
```

lemma unitaryConj_sub_smul_surjective [source]

If `A - z` is surjective for a scalar `z : ℂ`, then so is `u A u⁻¹ - z`.

```
lemma unitaryConj_sub_smul_surjective {z : ℂ} (h : Function.Surjective (A - z • 1).toFun) :
    Function.Surjective (unitaryConj u A - z • 1).toFun
```

lemma closure_le_adjoint [source]

```
lemma IsSymmetric.closure_le_adjoint [CompleteSpace H] (h : T.IsSymmetric) (h' : T.
    HasDenseDomain) :
    T.closure ≤ T†
```

lemma isEssentiallySelfAdjoint_iff [source]

```
lemma IsSymmetric.isEssentiallySelfAdjoint_iff [CompleteSpace H]
    (h : T.IsSymmetric) (h' : T.HasDenseDomain) :
    T.IsEssentiallySelfAdjoint ↔ T†.domain = T.closure.domain
```

lemma closure [source]

The closure of a symmetric densely-defined operator is symmetric: `T††` is a symmetric closed extension of `T`, so it extends `T.closure`, whose symmetry then descends.

```
lemma IsSymmetric.closure [CompleteSpace H] (hsym : T.IsSymmetric) (hdense : T.HasDenseDomain)
    :
    T.closure.IsSymmetric
```

lemma isSelfAdjoint_of_range_eq_top [source]

Self-adjointness from surjectivity of `T ± i`: a symmetric, densely-defined operator `T` for which `T + I • 1` and `T - I • 1` both have full range is self-adjoint.

```
lemma IsSymmetric.isSelfAdjoint_of_range_eq_top [CompleteSpace H] (hsym : T.IsSymmetric)
    (hdense : T.HasDenseDomain)
    (hadd : (T + I • 1).toFun.range = ⊤) (hsub : (T - I • 1).toFun.range = ⊤) :
    IsSelfAdjoint T
```

2.62 Physlib.QuantumMechanics.QuantumSystem.Basic

Physlib/QuantumMechanics/QuantumSystem/Basic.lean on GitHub.

structure QuantumSystem [source]

A quantum system is identified by its Hilbert space and self-adjoint Hamiltonian operator.

```
structure QuantumSystem where
  HS : Type*
  [instNormed : NormedAddCommGroup HS]
  [instInner : InnerProductSpace ℂ HS]
  [instComplete : CompleteSpace HS]
  ℋ : HS →ₗ.[ℂ] HS
  ℋ_self_adjoint : IsSelfAdjoint ℋ
```

instance NormedAddCommGroup Q.HS [source]

```
instance (Q : QuantumSystem) : NormedAddCommGroup Q.HS
```

instance InnerProductSpace ℂ Q.HS [source]

```
instance (Q : QuantumSystem) : InnerProductSpace ℂ Q.HS
```

instance CompleteSpace Q.HS [source]

```
instance (Q : QuantumSystem) : CompleteSpace Q.HS
```

def mk_esa [source]

Create a quantum system from a Hamiltonian operator which is merely essentially self-adjoint by taking its closure.

```
def mk_esa {HS : Type*} [NormedAddCommGroup HS] [InnerProductSpace ℂ HS] [CompleteSpace HS]
    {ℋ : HS →ₗ.[ℂ] HS} (hℋ : IsEssentiallySelfAdjoint ℋ) : QuantumSystem
```

instance instZero [source]

```
instance instZero : Zero QuantumSystem
```

def UnitaryRelation [source]

The relation on quantum systems where `Q₁` is related to `Q₂` if there exists a linear isometry equivalence `e : Q₁.HS ≃ₗᵢ[ℂ] Q₂.HS` between the respective Hilbert spaces satisfying `e ∘ Q₁.ℋ ∘ e.symm = Q₂.ℋ`.

```
def UnitaryRelation (Q₁ Q₂ : QuantumSystem) : Prop
```

lemma unitaryRelation_refl [source]

The relation `UnitaryRelation` is reflexive.

```
lemma unitaryRelation_refl (Q : QuantumSystem) : UnitaryRelation Q Q
```

lemma unitaryRelation_symm [source]

The relation `UnitaryRelation` is symmetric.

```
lemma unitaryRelation_symm {Q₁ Q₂ : QuantumSystem} (h₁₂ : UnitaryRelation Q₁ Q₂) :
    UnitaryRelation Q₂ Q₁
```

lemma unitaryRelation_trans [source]

The relation `UnitaryRelation` is transitive.

```
lemma unitaryRelation_trans {Q₁ Q₂ Q₃ : QuantumSystem} (h₁₂ : UnitaryRelation Q₁ Q₂)
    (h₂₃ : UnitaryRelation Q₂ Q₃) : UnitaryRelation Q₁ Q₃
```

lemma unitaryRelation_equiv [source]

The relation `UnitaryRelation` is an equivalence relation.

```
lemma unitaryRelation_equiv : Equivalence UnitaryRelation where
```

instance QuantumSystemSetoid [source]

The setoid of quantum systems with `UnitaryRelation` for equivalence relation.

```
instance QuantumSystemSetoid : Setoid QuantumSystem
```

2.63 Physlib.QuantumMechanics.RectangularBarrier.Basic

Physlib/QuantumMechanics/RectangularBarrier/Basic.lean on GitHub.

structure RectangularBarrier [source]

A quantum particle with mass `m > 0` on `Space 1` subject to a rectangular potential barrier.

The potential is `V₀` on the interval `Icc lower upper` and zero elsewhere.

```
structure RectangularBarrier where
  m : ℝ
  hm : 0 < m
  lower : ℝ
  upper : ℝ
  h_bounds : lower < upper
  V₀ : ℝ
```

lemma m_pos [source]

```
@[simp]
lemma m_pos : 0 < Q.m
```

lemma m_nonneg [source]

```
@[simp]
lemma m_nonneg : 0 ≤ Q.m
```

lemma m_ne_zero [source]

```
@[simp]
lemma m_ne_zero : Q.m ≠ 0
```

def potentialFunction [source]

The piece-wise constant potential, equal to `Q.V₀` for `x.val 0 ∈ Icc Q.lower Q.upper` and zero otherwise.

```
def potentialFunction : Space 1 → ℝ
```

lemma potentialFunction_eq [source]

```
lemma potentialFunction_eq :
    Q.potentialFunction = fun x ↦ (Icc Q.lower Q.upper).indicator (fun _ ↦ Q.V₀) (x 0)
```

informal_lemma potentialFunction_aestronglyMeasurable [source]

The piecewise-constant potential of the rectangular barrier is a.e. strongly measurable.

```
informal_lemma potentialFunction_aestronglyMeasurable where
  deps := [``RectangularBarrier]
  tag := "QM-RB-aesm"
```

abbrev HS [source]

The Hilbert space for the 1d rectangular barrier.

```
@[nolint unusedArguments]
abbrev HS (_ : RectangularBarrier) : Type _
```

def kineticOperator [source]

The kinetic energy operator, `p²/2m`.

```
def kineticOperator : Q.HS →ₗ.[ℂ] Q.HS
```

def potentialOperator [source]

The potential energy operator, defined by multiplication by `Q.potentialFunction`.

```
def potentialOperator : Q.HS →ₗ.[ℂ] Q.HS
```

informal_lemma potentialOperator_isSelfAdjoint [source]

The potential operator for the rectangular barrier is self-adjoint.

```
informal_lemma potentialOperator_isSelfAdjoint where
  deps := [``RectangularBarrier]
  tag := "QM-RB-sa"
```

informal_definition hamiltonian [source]

The Hamiltonian for the rectangular barrier.

```
informal_definition hamiltonian where
  deps := [``RectangularBarrier]
  tag := "QM-RB-ham"
```

informal_lemma hamiltonian_essentially_self_adjoint [source]

The Hamiltonian for the rectangular barrier is essentially self-adjoint.

```
informal_lemma hamiltonian_essentially_self_adjoint where
  deps := [``RectangularBarrier.hamiltonian]
  tag := "QM-RB-hamESA"
```

informal_definition toQuantumSystem [source]

The rectangular barrier as a quantum system (self-adjoint Hamiltonian acting on a Hilbert space).

```
informal_definition toQuantumSystem where
  deps := [``RectangularBarrier.hamiltonian_essentially_self_adjoint]
  tag := "QM-RB-sys"
```

2.64 Physlib.Relativity.Fermions.Dirac.Basic

Physlib/Relativity/Fermions/Dirac/Basic.lean on GitHub.

structure Dirac [source]

A Dirac fermion, consisting of a left handed Weyl fermion and a dual right handed Weyl fermion.

```
structure Dirac where
  left : LeftHandedWeyl
  dualRight : DualRightHandedWeyl
```

def decomposeEquiv [source]

The decomposition of a Dirac fermion into its left handed and dual right handed components.

```
def decomposeEquiv : Dirac ≃ LeftHandedWeyl × DualRightHandedWeyl where
```

instance AddCommGroup Dirac [source]

```
instance : AddCommGroup Dirac
```

instance Module ℂ Dirac [source]

```
instance : Module ℂ Dirac
```

lemma left_add [source]

```
@[simp]
lemma left_add (d₁ d₂ : Dirac) : (d₁ + d₂).left = d₁.left + d₂.left
```

lemma dualRight_add [source]

```
@[simp]
lemma dualRight_add (d₁ d₂ : Dirac) : (d₁ + d₂).dualRight = d₁.dualRight + d₂.dualRight
```

lemma left_smul [source]

```
@[simp]
lemma left_smul (c : ℂ) (d : Dirac) : (c • d).left = c • d.left
```

lemma dualRight_smul [source]

```
@[simp]
lemma dualRight_smul (c : ℂ) (d : Dirac) : (c • d).dualRight = c • d.dualRight
```

def decomposeLinEquiv [source]

The linear equivalence between `Dirac` and `LeftHandedWeyl × DualRightHandedWeyl`.

```
def decomposeLinEquiv : Dirac ≃ₗ[ℂ] LeftHandedWeyl × DualRightHandedWeyl where
```

def chiralBasis [source]

The chiral basis of the Dirac fermions.

```
def chiralBasis : Basis (Fin 4) ℂ Dirac
```

lemma chiralBasis_cast_add [source]

```
lemma chiralBasis_cast_add (i : Fin 2) : chiralBasis (Fin.castAdd 2 i) =
    ⟨LeftHandedWeyl.basis i, 0⟩
```

lemma chiralBasis_nat_add [source]

```
lemma chiralBasis_nat_add (i : Fin 2) : chiralBasis (Fin.natAdd 2 i) =
    ⟨0, DualRightHandedWeyl.basis i⟩
```

def rep [source]

The representation of `SL(2, ℂ)` on Dirac fermions.

```
def rep : Representation ℂ SL(2,ℂ) Dirac where
```

lemma rep_apply_mk [source]

```
lemma rep_apply_mk (g : SL(2,ℂ)) (ψ : LeftHandedWeyl) (χ : DualRightHandedWeyl) :
    rep g ⟨ψ, χ⟩ = ⟨LeftHandedWeyl.rep g ψ, DualRightHandedWeyl.rep g χ⟩
```

def decomposeRepEquiv [source]

The equivalence between the representation on `Dirac` and the representation on `LeftHandedWeyl × DualRightHandedWeyl`.

```
def decomposeRepEquiv : rep.Equiv ((LeftHandedWeyl.rep).prod (DualRightHandedWeyl.rep))  where
```

2.65 Physlib.Relativity.Fermions.Weyl.DualLeftHanded

Physlib/Relativity/Fermions/Weyl/DualLeftHanded.lean on GitHub.

lemma toFin2ℂ_eq_val [source]

```
lemma toFin2ℂ_eq_val (ψ : DualLeftHandedWeyl) : ψ.toFin2ℂ = ψ.val
```

def basis [source]

The standard basis on dual-left-handed Weyl fermions.

```
def basis : Basis (Fin 2) ℂ DualLeftHandedWeyl
```

lemma basis_apply [source]

```
lemma basis_apply (i j : Fin 2) : (basis i).1 j = if j = i then 1 else 0
```

lemma eq_sum_basis [source]

```
lemma eq_sum_basis (ψ : DualLeftHandedWeyl) : ψ = ∑ i, ψ.1 i • basis i
```

lemma basis_val [source]

```
lemma basis_val (i : Fin 2) : (basis i).val = Pi.single i 1
```

def rep [source]

The vector space ℂ^2 carrying the representation of SL(2,C) given by $M \to (M^{-1})^T$. In index notation corresponds to a left-handed Weyl fermion with indices ψ_a.

```
def rep : Representation ℂ SL(2,ℂ) DualLeftHandedWeyl where
```

lemma rep_apply_eq_sum_basis [source]

```
lemma rep_apply_eq_sum_basis (M : SL(2,ℂ)) (ψ : DualLeftHandedWeyl) :
    rep M ψ = ∑ i, (∑ j, M.1⁻¹ j i * ψ.1 j) • basis i
```

lemma rep_apply_basis [source]

```
lemma rep_apply_basis (M : SL(2,ℂ)) (i : Fin 2) :
    rep M (basis i) = ∑ j, M.1⁻¹ i j • basis j
```

lemma rep_toMatrix [source]

```
lemma rep_toMatrix (M : SL(2,ℂ)) : (LinearMap.toMatrix basis basis) (rep M) = (M.1⁻¹)ᵀ
```

lemma rep_apply_basis_repr [source]

```
lemma rep_apply_basis_repr (M : SL(2,ℂ)) (i j : Fin 2) :
    basis.repr (rep M (basis i)) j = M.1⁻¹ i j
```

2.66 Physlib.Relativity.Fermions.Weyl.DualRightHanded

Physlib/Relativity/Fermions/Weyl/DualRightHanded.lean on GitHub.

lemma toFin2ℂ_eq_val [source]

```
lemma toFin2ℂ_eq_val (ψ : DualRightHandedWeyl) : ψ.toFin2ℂ = ψ.val
```

def basis [source]

The standard basis on dual-right-handed Weyl fermions.

```
def basis : Basis (Fin 2) ℂ DualRightHandedWeyl
```

lemma basis_apply [source]

```
lemma basis_apply (i j : Fin 2) : (basis i).1 j = if j = i then 1 else 0
```

lemma eq_sum_basis [source]

```
lemma eq_sum_basis (ψ : DualRightHandedWeyl) : ψ = ∑ i, ψ.1 i • basis i
```

lemma basis_val [source]

```
lemma basis_val (i : Fin 2) : (basis i).val = Pi.single i 1
```

def rep [source]

The vector space ℂ^2 carrying the representation of SL(2,C) given by $M \to (M^{-1})^\dagger$.
In index notation this corresponds to a Weyl fermion with index `ψ_{dot a}`.

```
def rep : Representation ℂ SL(2,ℂ) DualRightHandedWeyl where
```

lemma rep_apply [source]

```
lemma rep_apply (M : SL(2,ℂ)) (ψ : DualRightHandedWeyl) :
    rep M ψ = ⟨(M.1⁻¹).conjTranspose *ᵥ ψ.1⟩
```

lemma rep_apply_eq_sum_basis [source]

```
lemma rep_apply_eq_sum_basis (M : SL(2,ℂ)) (ψ : DualRightHandedWeyl) :
    rep M ψ = ∑ i, (∑ j, (M.1⁻¹).conjTranspose i j * ψ.1 j) • basis i
```

lemma rep_apply_basis [source]

```
lemma rep_apply_basis (M : SL(2,ℂ)) (i : Fin 2) :
    rep M (basis i) = ∑ j, (M.1⁻¹).conjTranspose j i • basis j
```

lemma rep_toMatrix [source]

```
lemma rep_toMatrix (M : SL(2,ℂ)) :
    (LinearMap.toMatrix basis basis) (rep M) = (M.1⁻¹).conjTranspose
```

lemma rep_apply_basis_repr [source]

```
lemma rep_apply_basis_repr (M : SL(2,ℂ)) (i j : Fin 2) :
    basis.repr (rep M (basis i)) j = star (M.1⁻¹ i j)
```

2.67 Physlib.Relativity.Fermions.Weyl.Duals

Physlib/Relativity/Fermions/Weyl/Duals.lean on GitHub.

def LeftHandedWeyl.dual [source]

The morphism between the representation `leftHanded` and the representation `dualLeftHanded` defined by multiplying an element of `leftHanded` by the matrix `ε^{a₀a₁} = !![0, 1; -1, 0]`.

```
def LeftHandedWeyl.dual : LeftHandedWeyl.rep.IntertwiningMap DualLeftHandedWeyl.rep where
```

lemma LeftHandedWeyl.dual_hom_apply [source]

```
lemma LeftHandedWeyl.dual_hom_apply (ψ : LeftHandedWeyl) :
    LeftHandedWeyl.dual ψ =
    DualLeftHandedWeyl.toFin2ℂEquiv.symm (!![0, 1; -1, 0] *ᵥ ψ.toFin2ℂ)
```

def DualLeftHandedWeyl.dual [source]

The morphism from `dualLeftHanded` to `leftHanded` defined by multiplying an element of DualLeftHandedWeyl by the matrix `ε_{a₁a₂} = !![0, -1; 1, 0]`.

```
def DualLeftHandedWeyl.dual : DualLeftHandedWeyl.rep.IntertwiningMap LeftHandedWeyl.rep where
```

lemma DualLeftHandedWeyl.dual_hom_apply [source]

```
lemma DualLeftHandedWeyl.dual_hom_apply (ψ : DualLeftHandedWeyl) :
    DualLeftHandedWeyl.dual ψ =
    LeftHandedWeyl.toFin2ℂEquiv.symm (!![0, -1; 1, 0] *ᵥ ψ.toFin2ℂ)
```

def LeftHandedWeyl.dualEquiv [source]

The equivalence between the representation `leftHanded` and the representation `dualLeftHanded` defined by multiplying an element of `leftHanded` by the matrix `ε^{a₀a₁} = !![0, 1; -1, 0]`.

```
def LeftHandedWeyl.dualEquiv : LeftHandedWeyl.rep.Equiv DualLeftHandedWeyl.rep
```

lemma dualEquiv_hom_hom_apply [source]

`leftHandedDualEquiv` acting on an element `ψ : leftHanded` corresponds to multiplying `ψ` by the matrix `!![0, 1; -1, 0]`.

```
lemma LeftHandedWeyl.dualEquiv_hom_hom_apply (ψ : LeftHandedWeyl) :
    LeftHandedWeyl.dualEquiv ψ =
    DualLeftHandedWeyl.toFin2ℂEquiv.symm (!![0, 1; -1, 0] *ᵥ ψ.toFin2ℂ)
```

lemma dualEquiv_inv_hom_apply [source]

The inverse of `leftHandedDualEquiv` acting on an element`ψ : dualLeftHanded` corresponds to multiplying `ψ` by the matrix `!![0, -1; 1, 0]`.

```
lemma LeftHandedWeyl.dualEquiv_inv_hom_apply (ψ : DualLeftHandedWeyl) :
    LeftHandedWeyl.dualEquiv.symm ψ =
    LeftHandedWeyl.toFin2ℂEquiv.symm (!![0, -1; 1, 0] *ᵥ ψ.toFin2ℂ)
```

informal_definition RightHandedWeyl.dualEquiv [source]

The linear equivalence between `rightHandedWeyl` and `DualRightHandedWeyl` given by multiplying an element of `rightHandedWeyl` by the matrix `ε^{a a₁} = !![0, 1; -1, 0]`.

```
informal_definition RightHandedWeyl.dualEquiv where
  deps := [``RightHandedWeyl, ``DualRightHandedWeyl]
  tag := "6VZR4"
```

informal_lemma dualEquiv_equivariant [source]

The linear equivalence `rightHandedWeylDualEquiv` is equivariant with respect to the action of `SL(2,C)` on `rightHandedWeyl` and `DualRightHandedWeyl`.

```
informal_lemma RightHandedWeyl.dualEquiv_equivariant where
  deps := [``RightHandedWeyl.dualEquiv]
  tag := "6VZSG"
```

2.68 Physlib.Relativity.Fermions.Weyl.LeftHanded

Physlib/Relativity/Fermions/Weyl/LeftHanded.lean on GitHub.

lemma toFin2ℂ_eq_val [source]

```
lemma toFin2ℂ_eq_val (ψ : LeftHandedWeyl) : ψ.toFin2ℂ = ψ.val
```

def basis [source]

The standard basis on left-handed Weyl fermions.

```
def basis : Basis (Fin 2) ℂ LeftHandedWeyl
```

lemma basis_apply [source]

```
lemma basis_apply (i j : Fin 2) : (basis i).1 j = if j = i then 1 else 0
```

lemma eq_sum_basis [source]

```
lemma eq_sum_basis (ψ : LeftHandedWeyl) : ψ = ∑ i, ψ.1 i • basis i
```

lemma basis_val [source]

```
lemma basis_val (i : Fin 2) : (basis i).val = Pi.single i 1
```

def rep [source]

The vector space ℂ^2 carrying the fundamental representation of SL(2,C). In index notation corresponds to a Weyl fermion with indices ψ^a.

```
def rep : Representation ℂ SL(2,ℂ) LeftHandedWeyl where
```

lemma rep_apply [source]

```
lemma rep_apply (M : SL(2,ℂ)) (ψ : LeftHandedWeyl) : rep M ψ = ⟨M.1 *ᵥ ψ.1⟩
```

lemma rep_apply_eq_sum_basis [source]

```
lemma rep_apply_eq_sum_basis (M : SL(2,ℂ)) (ψ : LeftHandedWeyl) :
    rep M ψ = ∑ i, (∑ j, M.1 i j * ψ.1 j) • basis i
```

lemma rep_apply_basis [source]

```
lemma rep_apply_basis (M : SL(2,ℂ)) (i : Fin 2) :
    rep M (basis i) = ∑ j, M.1 j i • basis j
```

lemma rep_toMatrix [source]

```
lemma rep_toMatrix (M : SL(2,ℂ)) : (LinearMap.toMatrix basis basis) (rep M) = M.1
```

lemma rep_apply_basis_repr [source]

```
lemma rep_apply_basis_repr (M : SL(2,ℂ)) (i j : Fin 2) :
    basis.repr (rep M (basis i)) j = M.1 j i
```

2.69 Physlib.Relativity.Fermions.Weyl.RightHanded

Physlib/Relativity/Fermions/Weyl/RightHanded.lean on GitHub.

lemma toFin2ℂ_eq_val [source]

```
lemma toFin2ℂ_eq_val (ψ : RightHandedWeyl) : ψ.toFin2ℂ = ψ.val
```

def basis [source]

The standard basis on right-handed Weyl fermions.

```
def basis : Basis (Fin 2) ℂ RightHandedWeyl
```

lemma basis_apply [source]

```
lemma basis_apply (i j : Fin 2) : (basis i).1 j = if j = i then 1 else 0
```

lemma eq_sum_basis [source]

```
lemma eq_sum_basis (ψ : RightHandedWeyl) : ψ = ∑ i, ψ.1 i • basis i
```

lemma basis_val [source]

```
lemma basis_val (i : Fin 2) : (basis i).val = Pi.single i 1
```

def rep [source]

The vector space ℂ^2 carrying the conjugate representation of SL(2,C). In index notation corresponds to a Weyl fermion with indices ψ^{dot a}.

```
def rep : Representation ℂ SL(2,ℂ) RightHandedWeyl where
```

lemma rep_apply [source]

```
lemma rep_apply (M : SL(2,ℂ)) (ψ : RightHandedWeyl) : rep M ψ = ⟨M.1.map star *ᵥ ψ.1⟩
```

lemma rep_apply_eq_sum_basis [source]

```
lemma rep_apply_eq_sum_basis (M : SL(2,ℂ)) (ψ : RightHandedWeyl) :
    rep M ψ = ∑ i, (∑ j, M.1.map star i j * ψ.1 j) • basis i
```

lemma rep_apply_basis [source]

```
lemma rep_apply_basis (M : SL(2,ℂ)) (i : Fin 2) :
    rep M (basis i) = ∑ j, M.1.map star j i • basis j
```

lemma rep_toMatrix [source]

```
lemma rep_toMatrix (M : SL(2,ℂ)) : (LinearMap.toMatrix basis basis) (rep M) = M.1.map star
```

lemma rep_apply_basis_repr [source]

```
lemma rep_apply_basis_repr (M : SL(2,ℂ)) (i j : Fin 2) :
    basis.repr (rep M (basis i)) j = star (M.1 j i)
```

2.70 Physlib.Relativity.LorentzGroup.Basic

Physlib/Relativity/LorentzGroup/Basic.lean on GitHub.

lemma range_toGL [source]

The range of `toGL`, as a set in `GL (Fin 1 ⊕ Fin d) ℝ`, is exactly the preimage of the Lorentz group under the coercion `GL (Fin 1 ⊕ Fin d) ℝ → Matrix _ _ ℝ`: an element of the general linear group lies in the image precisely when its underlying matrix is a Lorentz transformation.

```
lemma range_toGL :
    Set.range (@toGL d).toFun =
      Units.val ⁻¹' (LorentzGroup d : Set (Matrix (Fin 1 ⊕ Fin d) (Fin 1 ⊕ Fin d) ℝ))
```

lemma toGL_isClosedEmbedding [source]

The homomorphism `toGL` is a closed embedding of the Lorentz group into `GL (Fin 1 ⊕ Fin d) ℝ`. This strengthens `toGL_embedding`: the image is closed because it is the preimage, under the continuous coercion `GL (Fin 1 ⊕ Fin d) ℝ → Matrix _ _ ℝ`, of the closed set `LorentzGroup d` (`LorentzGroup.isClosed`).

```
lemma toGL_isClosedEmbedding : Topology.IsClosedEmbedding (@toGL d).toFun where
```

def toGLSubgroup [source]

The Lorentz group realised as a subgroup of `GL (Fin 1 ⊕ Fin d) ℝ`, namely the range of the embedding `toGL`.

```
def toGLSubgroup (d : ℕ) : Subgroup (GL (Fin 1 ⊕ Fin d) ℝ)
```

lemma mem_toGLSubgroup [source]

```
@[simp]
lemma mem_toGLSubgroup {x : GL (Fin 1 ⊕ Fin d) ℝ} :
    x ∈ toGLSubgroup d ↔
      (x : Matrix (Fin 1 ⊕ Fin d) (Fin 1 ⊕ Fin d) ℝ) ∈ LorentzGroup d
```

lemma coe_toGLSubgroup [source]

```
lemma coe_toGLSubgroup :
    (toGLSubgroup d : Set (GL (Fin 1 ⊕ Fin d) ℝ)) =
      Units.val ⁻¹' (LorentzGroup d : Set (Matrix (Fin 1 ⊕ Fin d) (Fin 1 ⊕ Fin d) ℝ))
```

def toClosedSubgroup [source]

*The Lorentz group realised as a *closed* subgroup of `GL (Fin 1 ⊕ Fin d) ℝ`. The closedness is exactly the content of `toGL_isClosedEmbedding`, and is the hypothesis that the (not-yet-formalised) closed subgroup theorem would consume to produce a Lie group structure.*

```
def toClosedSubgroup (d : ℕ) : ClosedSubgroup (GL (Fin 1 ⊕ Fin d) ℝ) where
```

2.71 Physlib.Relativity.LorentzGroup.Restricted.FromBoostRotation

Physlib/Relativity/LorentzGroup/Restricted/FromBoostRotation.lean on GitHub.

theorem exists_boost_mul_rotation [source]

Every element of the restricted Lorentz group can be written as a product of a generalized boost and a rotation.

```
theorem exists_boost_mul_rotation {d} (Λ : LorentzGroup.restricted d) :
    ∃ (v : Lorentz.Velocity d) (R : Rotations d),
      (Λ : LorentzGroup d) = generalizedBoost 0 v * (R : LorentzGroup d)
```

2.72 Physlib.Relativity.Tensors.Basic

Physlib/Relativity/Tensors/Basic.lean on GitHub.

lemma permT_injective [source]

permT is injective.

```
lemma permT_injective {n m : ℕ} {c : Fin n → C} {c1 : Fin m → C}
    {σ : Fin m → Fin n} (h : IsReindexing c c1 σ) :
    Function.Injective (permT (S := S) σ h)
```

2.73 Physlib.Relativity.Tensors.ComplexTensor.Metrics.Basic

Physlib/Relativity/Tensors/ComplexTensor/Metrics/Basic.lean on GitHub.

lemma leftMetric_eq_leftHandedWeyl_basis [source]

```
lemma leftMetric_eq_leftHandedWeyl_basis : εL =
    - fromPairT (LeftHandedWeyl.basis 0 ⊗ₜ[ℂ] LeftHandedWeyl.basis 1)
    + fromPairT (LeftHandedWeyl.basis 1 ⊗ₜ[ℂ] LeftHandedWeyl.basis 0)
```

lemma dualLeftMetric_eq_dualLeftHandedWeyl_basis [source]

```
lemma dualLeftMetric_eq_dualLeftHandedWeyl_basis : εL' =
    fromPairT (DualLeftHandedWeyl.basis 0 ⊗ₜ[ℂ] DualLeftHandedWeyl.basis 1)
    - fromPairT (DualLeftHandedWeyl.basis 1 ⊗ₜ[ℂ] DualLeftHandedWeyl.basis 0)
```

lemma rightMetric_eq_rightHandedWeyl_basis [source]

```
lemma rightMetric_eq_rightHandedWeyl_basis : εR =
    - fromPairT (RightHandedWeyl.basis 0 ⊗ₜ[ℂ] RightHandedWeyl.basis 1)
    + fromPairT (RightHandedWeyl.basis 1 ⊗ₜ[ℂ] RightHandedWeyl.basis 0)
```

lemma dualRightMetric_eq_dualRightHandedWeyl_basis [source]

```
lemma dualRightMetric_eq_dualRightHandedWeyl_basis : εR' =
    fromPairT (DualRightHandedWeyl.basis 0 ⊗ₜ[ℂ] DualRightHandedWeyl.basis 1)
    - fromPairT (DualRightHandedWeyl.basis 1 ⊗ₜ[ℂ] DualRightHandedWeyl.basis 0)
```

2.74 Physlib.Relativity.Tensors.ComplexTensor.Units.Basic

Physlib/Relativity/Tensors/ComplexTensor/Units/Basic.lean on GitHub.

lemma dualLeftLeftUnit_eq_tensor_basis [source]

```
lemma dualLeftLeftUnit_eq_tensor_basis : δL' =
    ∑ i, Tensor.basis (S := complexLorentzTensor)
      ![Color.downL, Color.upL] (fun | 0 => i | 1 => i)
```

lemma leftDualLeftUnit_eq_tensor_basis [source]

```
lemma leftDualLeftUnit_eq_tensor_basis : δL =
    ∑ i, Tensor.basis (S := complexLorentzTensor)
      ![Color.upL, Color.downL] (fun | 0 => i | 1 => i)
```

lemma dualRightRightUnit_eq_tensor_basis [source]

```
lemma dualRightRightUnit_eq_tensor_basis : δR' =
    ∑ i, Tensor.basis (S := complexLorentzTensor)
      ![Color.downR, Color.upR] (fun | 0 => i | 1 => i)
```

lemma rightDualRightUnit_eq_tensor_basis [source]

```
lemma rightDualRightUnit_eq_tensor_basis : δR =
    ∑ i, Tensor.basis (S := complexLorentzTensor)
      ![Color.upR, Color.downR] (fun | 0 => i | 1 => i)
```

2.75 Physlib.Relativity.Tensors.Conjugation.Basic

Physlib/Relativity/Tensors/Conjugation/Basic.lean on GitHub.

lemma conjEquiv_basis_repr [source]

```
lemma conjEquiv_basis_repr {c : C} (v : V c) (φ : basisIdx (S.bar c)) :
    (b (S.bar c)).repr (S.conjEquiv v) φ
      = star ((b c).repr v (Equiv.cast (S.barIdx_eq c) φ))
```

lemma conjEquiv_equivariant [source]

```
lemma conjEquiv_equivariant {c : C} (g : G) (v : V c) :
    S.conjEquiv (rep c g v) = rep (S.bar c) g (S.conjEquiv v)
```

lemma conjT_apply [source]

```
lemma conjT_apply {n : ℕ} {c : Fin n → C} (t : S.Tensor c) :
    S.conjT t = ofComponents (fun i => S.bar (c i))
      (fun b => star (componentMap c t (S.componentReindex c b)))
```

lemma conjT_basis [source]

```
lemma conjT_basis {n : ℕ} {c : Fin n → C} (i : ComponentIdx (S := S.toTensorSpecies) c) :
    S.conjT (basis c i) = basis (fun i => S.bar (c i)) ((S.componentReindex c).symm i)
```

lemma conjT_pure [source]

```
lemma conjT_pure {n : ℕ} {c : Fin n → C} (p : Tensor.Pure S.toTensorSpecies c) :
    S.conjT p.toTensor = Pure.toTensor (fun i => S.conjEquiv (p i))
```

lemma conjT_equivariant [source]

```
lemma conjT_equivariant {n : ℕ} {c : Fin n → C} (g : G) (t : S.Tensor c) :
    S.conjT (g • t) = g • S.conjT t
```

2.76 Physlib.Relativity.Tensors.Contraction.CrossToEnd

Physlib/Relativity/Tensors/Contraction/CrossToEnd.lean on GitHub.

def crossToEnd [source]

Slot-addressed cross contraction: contract slot `i` of `t1` against slot `j` of `t2`. The survivors are `t1`'s slots except `i` followed by `t2`'s slots except `j`, so the output color is a clean `Fin (nA + nB) → C` and a chain composes without recasts.

```
noncomputable def crossToEnd {nA nB : ℕ} {cA : Fin (nA + 1) → C} {cB : Fin (nB + 1) → C}
    (i : Fin (nA + 1)) (j : Fin (nB + 1)) (hc : S.τ (cA i) = cB j) :
    Tensor S cA →[k] Tensor S cB →[k]
      Tensor S (Fin.append (cA ∘ i.succAbove) (cB ∘ j.succAbove))
```

lemma crossToEnd_two [source]

Cross-contracting the last slot of a rank-2 tensor `A` with the `0`-slot of a rank-2 tensor `B` is a plain `contrT` of their product on the middle slots `1, 2`, up to a color recast. Discharging the slot arithmetic once lets a `contrT`-level identity lift to `crossToEnd` by one rewrite.

```
lemma crossToEnd_two {cA cB : Fin 2 → C} (h : S.τ (cA (Fin.last 1)) = cB 0)
    (A : Tensor S cA) (B : Tensor S cB) :
    crossToEnd (Fin.last 1) (0 : Fin 2) h A B
      = permT (id : Fin 2 → Fin 2) (IsReindexing.on_id.mpr (fun i => by fin_cases i <;> rfl))
          (contrT 2 (1 : Fin 4) (2 : Fin 4)
            (by
              refine (by decide, ?_)
              rw [show (1 : Fin 4) = Fin.castAdd 2 (1 : Fin 2) from rfl,
                  show (2 : Fin 4) = Fin.natAdd 2 (0 : Fin 2) from rfl,
                  Fin.append_left, Fin.append_right]
              exact h)
            (prodT A B))
```

lemma crossToEnd_equivariant [source]

Cross contraction is `G`-equivariant, each of `prodT`, `contrT` and `permT` being so.

```
@[simp]
lemma crossToEnd_equivariant {nA nB : ℕ} {cA : Fin (nA + 1) → C} {cB : Fin (nB + 1) → C}
    (i : Fin (nA + 1)) (j : Fin (nB + 1)) (hc : S.τ (cA i) = cB j) (g : G)
    (t1 : Tensor S cA) (t2 : Tensor S cB) :
    crossToEnd i j hc (g • t1) (g • t2) = g • crossToEnd i j hc t1 t2
```

lemma crossToEnd_assoc_rankTwo [source]

Rebracket the chain `A −(iA·0)− B −(last·0)− C` for rank-two `B` and `C`: contracting `A·B` first and then `·C` equals contracting `B·C` first and then `A·`, up to the color recast `permT id`.

Not full associativity: only `A`'s attachment slot `iA` is free. `B`'s rank two is what makes the statement possible, the second contraction being addressed at `Fin.last nA`, the slot `B`'s unique survivor occupies in `A·B`. Other slots are reached by conjugating with the `permT` commutators.

With `A := T, B := g, C := g⁻¹` this reassociates `(T·g)·g⁻¹` to `T·(g·g⁻¹)`, after which `g·g⁻¹` collapses to the unit tensor: the step behind raising then lowering a named index.

```
lemma crossToEnd_assoc_rankTwo {nA : ℕ}
    {cA : Fin (nA + 1) → C} {cB cC : Fin 2 → C}
    (iA : Fin (nA + 1))
    (hc1 : S.τ (cA iA) = cB 0)
    (hc2 : S.τ (cB (Fin.last 1)) = cC 0)
    (tA : Tensor S cA) (tB : Tensor S cB) (tC : Tensor S cC) :
    crossToEnd (Fin.last nA) 0
        (by
          rw [show (Fin.last nA : Fin (nA + 1)) = Fin.natAdd nA (Fin.last 0) from
              Fin.ext (by simp), Fin.append_right, Function.comp_apply, Fin.succAbove_zero,
            Fin.succ_last]
          exact hc2)
        (crossToEnd iA 0 hc1 tA tB) tC =
      permT id
        (IsReindexing.on_id.mpr fun i => by
          refine Fin.addCases (fun a => ?_) (fun a => ?_) i
          · simp only [Fin.append_left, Function.comp_apply, Fin.succAbove_last,
              show a.castSucc = Fin.castAdd 1 a from rfl, Fin.append_left]
          · simp only [Fin.append_right, Function.comp_apply, Fin.succAbove_zero]
            fin_cases a
            rfl)
        (crossToEnd iA 0
          (by
            rw [show (0 : Fin (1 + 1)) = Fin.castAdd 1 (0 : Fin 1) from rfl, Fin.append_left,
              Function.comp_apply, Fin.succAbove_last, Fin.castSucc_zero]
            exact hc1)
          tA
          (crossToEnd (Fin.last 1) 0 hc2 tB tC))
```

lemma crossToEnd_permT_left [source]

Left commutator: contracting slot `i` of `permT σ t1` equals contracting slot `σ i` of `t1`, then relabelling the survivors by `σ'`. The square `hσ'` records how `σ'` carries the complement of `i` to the complement of `σ i`, and implies its bijectivity.

```
lemma crossToEnd_permT_left {nA nB : ℕ} {cA cA' : Fin (nA + 1) → C}
    {cB : Fin (nB + 1) → C}
    (i : Fin (nA + 1)) (j : Fin (nB + 1))
    (σ : Fin (nA + 1) → Fin (nA + 1)) (σ' : Fin nA → Fin nA)
    (hσ : IsReindexing cA cA' σ)
    (hσ' : (σ i).succAbove ∘ σ' = σ ∘ i.succAbove)
    (hc : S.τ (cA' i) = cB j)
    (t1 : Tensor S cA) (t2 : Tensor S cB) :
    crossToEnd i j hc (permT σ hσ t1) t2 =
      permT (Fin.append (Fin.castAdd nB ∘ σ') (Fin.natAdd nA))
        (IsReindexing.append_congr_left (cB ∘ j.succAbove)
          (IsReindexing.succAbove_of_succAbove_eq i hσ hσ'))
        (crossToEnd (σ i) j (by rw [hσ.2 i]; exact hc) t1 t2)
```

lemma crossToEnd_permT_right [source]

Right commutator, the mirror of `crossToEnd_permT_left`: contracting slot `j` of `permT σ t2` equals contracting slot `σ j` of `t2`, then relabelling the survivors by `σ'`.

```
lemma crossToEnd_permT_right {nA nB : ℕ} {cA : Fin (nA + 1) → C}
    {cB cB' : Fin (nB + 1) → C}
```

```
    (i : Fin (nA + 1)) (j : Fin (nB + 1))
    (σ : Fin (nB + 1) → Fin (nB + 1)) (σ' : Fin nB → Fin nB)
    (hσ : IsReindexing cB cB' σ)
    (hσ' : (σ j).succAbove ∘ σ' = σ ∘ j.succAbove)
    (hc : S.τ (cA i) = cB' j)
    (t1 : Tensor S cA) (t2 : Tensor S cB) :
    crossToEnd i j hc t1 (permT σ hσ t2) =
      permT (Fin.append (Fin.castAdd nB) (Fin.natAdd nA ∘ σ'))
        (IsReindexing.append_congr_right (cA ∘ i.succAbove)
          (IsReindexing.succAbove_of_succAbove_eq j hσ hσ'))
        (crossToEnd i (σ j) (by rw [hσ.2 j]; exact hc) t1 t2)
```

2.77 Physlib.Relativity.Tensors.Contraction.CrossToSlot

Physlib/Relativity/Tensors/Contraction/CrossToSlot.lean on GitHub.

lemma crossToSlot_cycle [source]

The survivor color of `crossToEnd i j` against a rank-2 tensor of color `cM` is `c` with slot `i` replaced by the surviving color `cM (j.succAbove 0)`, once the survivors are rotated back by the inverse of the cycle `[i, last]`.

```
lemma IsReindexing.crossToSlot_cycle {nA : ℕ} {c : Fin (nA + 1) → C} {cM : Fin 2 → C}
    (i : Fin (nA + 1)) (j : Fin 2) :
    IsReindexing
      (Fin.append (c ∘ i.succAbove) (cM ∘ j.succAbove))
      (Function.update c i (cM (j.succAbove 0)))
      ⇑(Fin.cycleIcc i (Fin.last nA)).symm
```

def crossToSlot [source]

Contract slot `i` of `t` against slot `j` of the rank-2 tensor `M`, whose `j`-slot color must equal the dual color `τ (c i)` (propositionally, via `hc`), and rotate `M`'s surviving slot back into position `i`. The result is `t` with slot `i` relabelled to `M`'s surviving color.

This is raising and lowering a named index; the metric case is `j = 0, M = metricTensor (τ (c i))`. Because `hc` is propositional, a rank-2 tensor whose slot color matches only up to an equality enters directly, with no transport.

```
noncomputable def crossToSlot {nA : ℕ} {c : Fin (nA + 1) → C} {cM : Fin 2 → C}
    (i : Fin (nA + 1)) (j : Fin 2) (hc : S.τ (c i) = cM j) (M : S.Tensor cM) :
    S.Tensor c →[k] S.Tensor (Function.update c i (cM (j.succAbove 0)))
```

lemma crossToSlot_eq_crossToEnd [source]

The bridge between the two conventions: `crossToSlot i j hc M t` is `crossToEnd i j hc t M` with the survivor rotated from the last slot back to slot `i` by the inverse cycle.

```
lemma crossToSlot_eq_crossToEnd {nA : ℕ} {c : Fin (nA + 1) → C} {cM : Fin 2 → C}
    (i : Fin (nA + 1)) (j : Fin 2) (hc : S.τ (c i) = cM j) (M : S.Tensor cM) (t : S.Tensor c) :
    crossToSlot i j hc M t =
      permT ⇑(Fin.cycleIcc i (Fin.last nA)).symm (IsReindexing.crossToSlot_cycle i j)
        (crossToEnd i j hc t M)
```

def crossToSlotInv [source]

Contract slot `i` of a tensor whose color there is `d` against `M'`, then absorb the color cast the two `Function.update`s generate, landing back on `c`. This is the returning half of a raise-then-lower round trip; absorbing the cast here is what keeps the round trip cast-free at both ends.

```
noncomputable def crossToSlotInv {nA : ℕ} {c : Fin (nA + 1) → C} {b d e : C} (i : Fin (nA + 1))
    (he : c i = e) (hb : S.τ d = b) (M' : S.Tensor ![b, e]) :
    S.Tensor (Function.update c i d) →ₗ[k] S.Tensor c
```

lemma crossToSlot_permT_right_id [source]

An identity reindexing of the rank-2 tensor becomes the corresponding identity reindexing of the contracted output: the `crossToSlot`-level case of `crossToEnd_permT_right` where the rank-2 tensor's colors are only propositionally recast.

```
lemma crossToSlot_permT_right_id {nA : ℕ} {c : Fin (nA + 1) → C} {cM cM' : Fin 2 → C}
    (i : Fin (nA + 1)) (j : Fin 2) (hc : S.τ (c i) = cM' j)
    (M : Tensor S cM) (hM : IsReindexing cM cM' (id : Fin 2 → Fin 2)) (t : Tensor S c) :
    crossToSlot i j hc (permT (id : Fin 2 → Fin 2) hM M) t =
      permT (id : Fin (nA + 1) → Fin (nA + 1))
        (IsReindexing.on_id.mpr (fun a => by
          by_cases ha : a = i
          · subst ha
            simpa using hM.2 (j.succAbove 0)
          · simp [Function.update_of_ne ha]))
        (crossToSlot i j (hc.trans (hM.2 j).symm) M t)
```

lemma crossToSlot_equivariant [source]

Cross contraction into a slot is `G`-equivariant, both constituents (`crossToEnd`, `permT`) being so.

```
@[simp]
lemma crossToSlot_equivariant {nA : ℕ} {c : Fin (nA + 1) → C} {cM : Fin 2 → C}
    (i : Fin (nA + 1)) (j : Fin 2) (hc : S.τ (c i) = cM j) (g : G)
    (M : Tensor S cM) (t : Tensor S c) :
    crossToSlot i j hc (g • M) (g • t) = g • crossToSlot i j hc M t
```

2.78 Physlib.Relativity.Tensors.Contraction.Products

Physlib/Relativity/Tensors/Contraction/Products.lean on GitHub.

lemma prodT_contrT_fst [source]

A contraction internal to the left factor commutes past the outer product with a right spectator. The mirror of `prodT_contrT_snd`, obtained from it by `prodT_swap`, so the right-hand side contracts the swapped product `prodT t1 t` and carries a block swap.

```
lemma prodT_contrT_fst {n n1 : ℕ} {c : Fin (n + 1 + 1) → C}
    {c1 : Fin n1 → C}
    (i j : Fin (n + 1 + 1)) (hij : i ≠ j ∧ S.τ (c i) = c j)
    (t : Tensor S c) (t1 : Tensor S c1) :
    prodT (contrT n i j hij t) t1 =
    permT _ IsReindexing.append_swap
```

```
    (permT id (IsReindexing.append_succSuccAbove_natAdd i j) <|
    contrT _
      (Fin.natAdd n1 i)
      (Fin.natAdd n1 j)
      (by simpa using hij) <|
    prodT t1 t)
```

2.79 Physlib.Relativity.Tensors.Contraction.Pure

Physlib/Relativity/Tensors/Contraction/Pure.lean on GitHub.

lemma permP_dropPair_dropPair_congr [source]

Two `permP`–`dropPair`–`dropPair` towers over a common pure tensor agree once their composite slot maps agree pointwise. Comparing two orders of dropping two pairs of slots this way keeps every slot map an opaque variable, so the comparison closes by unification rather than by `whnf`-reducing the color and index-map composites.

```
lemma permP_dropPair_dropPair_congr {nP nOut : ℕ}
    {c : Fin (nP + 1 + 1 + 1 + 1) → C} {cOut : Fin nOut → C} (p : Pure S c)
    (aL bL : Fin (nP + 1 + 1 + 1 + 1)) (habL : aL ≠ bL)
    (a2L b2L : Fin (nP + 1 + 1)) (hab2L : a2L ≠ b2L)
    (σ1L : Fin nOut → Fin nP)
    (h1L : IsReindexing ((c ∘ Fin.succSuccAbove aL bL) ∘ Fin.succSuccAbove a2L b2L) cOut σ1L)
    (aR bR : Fin (nP + 1 + 1 + 1 + 1)) (habR : aR ≠ bR)
    (a2R b2R : Fin (nP + 1 + 1)) (hab2R : a2R ≠ b2R)
    (σ1R : Fin nOut → Fin nP)
    (h1R : IsReindexing ((c ∘ Fin.succSuccAbove aR bR) ∘ Fin.succSuccAbove a2R b2R) cOut σ1R)
    (hslot : ∀ m : Fin nOut,
      Fin.succSuccAbove aL bL (Fin.succSuccAbove a2L b2L (σ1L m)) =
      Fin.succSuccAbove aR bR (Fin.succSuccAbove a2R b2R (σ1R m))) :
    permP σ1L h1L (dropPair a2L b2L hab2L (dropPair aL bL habL p)) =
    permP σ1R h1R (dropPair a2R b2R hab2R (dropPair aR bR habR p))
```

lemma permP_dropPair_permP_congr [source]

Two `permP`–`dropPair`–`permP` towers over a common pure tensor agree once their composite slot maps agree pointwise. The one-contraction, inner-relabelled sibling of `permP_dropPair_dropPair_congr`.

```
lemma permP_dropPair_permP_congr {nP nOut mL mR : ℕ}
    {c : Fin nP → C} {cL : Fin (mL + 1 + 1) → C} {cR : Fin (mR + 1 + 1) → C}
    {cOut : Fin nOut → C} (p : Pure S c)
    (σ0L : Fin (mL + 1 + 1) → Fin nP) (h0L : IsReindexing c cL σ0L)
    (aL bL : Fin (mL + 1 + 1)) (habL : aL ≠ bL)
    (σ1L : Fin nOut → Fin mL)
    (h1L : IsReindexing (cL ∘ Fin.succSuccAbove aL bL) cOut σ1L)
    (σ0R : Fin (mR + 1 + 1) → Fin nP) (h0R : IsReindexing c cR σ0R)
    (aR bR : Fin (mR + 1 + 1)) (habR : aR ≠ bR)
    (σ1R : Fin nOut → Fin mR)
    (h1R : IsReindexing (cR ∘ Fin.succSuccAbove aR bR) cOut σ1R)
    (hslot : ∀ m : Fin nOut,
      σ0L (Fin.succSuccAbove aL bL (σ1L m)) = σ0R (Fin.succSuccAbove aR bR (σ1R m))) :
    permP σ1L h1L (dropPair aL bL habL (permP σ0L h0L p)) =
    permP σ1R h1R (dropPair aR bR habR (permP σ0R h0R p))
```

lemma contrPCoeff_congr [source]

contrPCoeff is insensitive to its proof arguments once the two contracted slots agree. Lets a coefficient comparison rewrite the slots without a congr search through the proof fields.

```
lemma contrPCoeff_congr {n : ℕ} {c : Fin n → C} (p : Pure S c)
    {i i' j j' : Fin n} (hi : i = i') (hj : j = j')
    {hij : i ≠ j ∧ S.τ (c i) = c j} {hij' : i' ≠ j' ∧ S.τ (c i') = c j'} :
    p.contrPCoeff i j hij = p.contrPCoeff i' j' hij'
```

2.80 Physlib.Relativity.Tensors.Contraction.SuccSuccAbove

Physlib/Relativity/Tensors/Contraction/SuccSuccAbove.lean on GitHub.

lemma succSuccAbove_castAdd_natAdd_apply_castAdd [source]

Reinserting a left-block survivor a after removing the i-th slot of the left block and the j-th slot of the right block of Fin ((nA + 1) + (nB + 1)), the removal read at the contracted length (nA + nB) + 1 + 1. Unlike succSuccAbove_natAdd_apply_castAdd the two holes straddle the two blocks, so the statement carries the reshaping Fin.casts.

```
lemma succSuccAbove_castAdd_natAdd_apply_castAdd {nA nB : ℕ} (i : Fin (nA + 1)) (j : Fin (nB +
    1))
    (a : Fin nA) :
    Fin.cast (show (nA + nB) + 1 + 1 = (nA + 1) + (nB + 1) by omega)
        ((Fin.cast (show (nA + 1) + (nB + 1) = (nA + nB) + 1 + 1 by omega)
            (Fin.castAdd (nB + 1) i)).succSuccAbove
          (Fin.cast (show (nA + 1) + (nB + 1) = (nA + nB) + 1 + 1 by omega)
            (Fin.natAdd (nA + 1) j)) (Fin.castAdd nB a))
      = Fin.castAdd (nB + 1) (i.succAbove a)
```

lemma succSuccAbove_castAdd_natAdd_apply_natAdd [source]

Reinserting a right-block survivor, the mirror of Fin.succSuccAbove_castAdd_natAdd_apply_castAdd.

```
lemma succSuccAbove_castAdd_natAdd_apply_natAdd {nA nB : ℕ} (i : Fin (nA + 1)) (j : Fin (nB +
    1))
    (a : Fin nB) :
    Fin.cast (show (nA + nB) + 1 + 1 = (nA + 1) + (nB + 1) by omega)
        ((Fin.cast (show (nA + 1) + (nB + 1) = (nA + nB) + 1 + 1 by omega)
            (Fin.castAdd (nB + 1) i)).succSuccAbove
          (Fin.cast (show (nA + 1) + (nB + 1) = (nA + nB) + 1 + 1 by omega)
            (Fin.natAdd (nA + 1) j)) (Fin.natAdd nA a))
      = Fin.natAdd (nA + 1) (j.succAbove a)
```

2.81 Physlib.Relativity.Tensors.Contraction.UnitTensorContraction

Physlib/Relativity/Tensors/Contraction/UnitTensorContraction.lean on GitHub.

lemma crossToEnd_prodT_unitTensor [source]

Cross-contracting the last slot of the product E ⊗ B (with B rank one) against the unit tensor for that slot's color returns E ⊗ B unchanged, the contracted slot carried to the end. The rank-one spectator case that seeds crossToEnd_unitTensor_slot.

```
private lemma crossToEnd_prodT_unitTensor {nV : ℕ} {cE : Fin nV → C} {cB : C}
    (E : Tensor S cE) (B : Tensor S ![cB]) :
    crossToEnd (Fin.last nV) (0 : Fin 2)
        (by rw [show (Fin.last nV : Fin (nV + 1)) = Fin.natAdd nV 0 from Fin.ext (by simp)];
    simp :
          S.τ ((Fin.append cE ![cB]) (Fin.last nV)) = ![S.τ cB, cB] 0)
        (prodT E B) (unitTensor cB) =
      permT id (IsReindexing.on_id.mpr (fun i => by
        refine Fin.addCases (fun j => ?_) (fun j => ?_) i
        · simp only [Fin.append_left, Function.comp_apply, Fin.succAbove_last]
          exact (Fin.append_left cE ![cB] j).symm
        · simp only [Fin.append_right, Function.comp_apply, Fin.succAbove_zero]
          fin_cases j
          rfl)) (prodT E B)
```

lemma crossToEnd_unitTensor_slot [source]

The boundary case of `crossToEnd_unitTensor`, proved by spectator decomposition along the last slot (`eq_sum_evalT`). The last slot is threaded as a variable `i` with `hilast : i = last nA` so that `crossToEnd_unitTensor` can apply it at the image of `i` under its transposition without a dependent rewrite of the slot index.

```
private lemma crossToEnd_unitTensor_slot {nA : ℕ} {cA : Fin (nA + 1) → C} {c : C}
    {i : Fin (nA + 1)} (hilast : i = Fin.last nA) (hc : cA i = c) (t1 : Tensor S cA) :
    crossToEnd i (0 : Fin 2) (by simp [hc] : S.τ (cA i) = ![S.τ c, c] 0) t1 (unitTensor c) =
      permT (Fin.append i.succAbove (fun _ : Fin 1 => i)) (IsReindexing.move_last i hc) t1
```

lemma crossToEnd_unitTensor [source]

Contracting slot `i` of `t1` against the unit tensor for that slot's color returns `t1`, the contracted slot relabelled by `move_last i`. The unit tensor acts as an identity for `crossToEnd` at any named slot.

```
lemma crossToEnd_unitTensor {nA : ℕ} {cA : Fin (nA + 1) → C} {c : C}
    (i : Fin (nA + 1)) (hc : cA i = c) (t1 : Tensor S cA) :
    crossToEnd i (0 : Fin 2) (by simp [hc] : S.τ (cA i) = ![S.τ c, c] 0) t1 (unitTensor c) =
      permT (Fin.append i.succAbove (fun _ : Fin 1 => i)) (IsReindexing.move_last i hc) t1
```

lemma unitTensor_pair [source]

The color list `crossToEnd` produces from a matched rank-2 pair `M : ![a, d]`, `M' : ![b, e]` is the color list `![S.τ e, e]` of `unitTensor e`, via the identity slot map: the reindexing carried by the collapse hypothesis `M · M' = δ` of the round trips below. Only the surviving colors `a` and `e` enter, and `a` equals `S.τ e` only propositionally, so a pair whose colors match propositionally enters with no transport.

```
lemma IsReindexing.unitTensor_pair {a b d e : C} (h : S.τ e = a) :
    IsReindexing (![S.τ e, e] : Fin 2 → C)
      (Fin.append ((![a, d] : Fin 2 → C) ∘ (Fin.last 1).succAbove)
        ((![b, e] : Fin 2 → C) ∘ (0 : Fin 2).succAbove))
      (id : Fin 2 → Fin 2)
```

lemma crossToEnd_round_trip_of_unit_slot [source]

Round trip of two contractions against a matched pair, in result-to-end form. Contracting slot `i` of `t` against `M₁`, then contracting the surviving `M₁`-index (deposited at slot `j`) against `M₂`, returns `t` up to the cycle that moves the round-tripped slot from the end back to `i`, whenever `M₁` and `M₂` collapse to the unit tensor. The last slot is threaded as a variable `j` with `hj : j = last nA` so that a caller can apply it at a propositionally-last slot without a dependent rewrite of the slot index.

```
lemma crossToEnd_round_trip_of_unit_slot {nA : ℕ} {c : Fin (nA + 1) → C} {d : C}
    (i j : Fin (nA + 1)) (hj : j = Fin.last nA)
    (M₁ : Tensor S ![S.τ (c i), d]) (M₂ : Tensor S ![S.τ d, c i])
    (hM : crossToEnd (Fin.last 1) (0 : Fin 2) rfl M₁ M₂ =
      permT (id : Fin 2 → Fin 2) (IsReindexing.unitTensor_pair rfl)
        (unitTensor (S := S) (c i)))
    (t : Tensor S c) :
    crossToEnd j (0 : Fin 2)
        (by subst hj
            rw [show (Fin.last nA : Fin (nA + 1)) = Fin.natAdd nA (0 : Fin 1) from
                  Fin.ext (by simp), Fin.append_right]; rfl)
        (crossToEnd i (0 : Fin 2) rfl t M₁) M₂ =
      permT (Fin.append i.succAbove (fun _ : Fin 1 => i))
        (by
          subst hj
          exact ⟨by
            rw [Fin.append_succAbove_const_eq_cycleIcc i]
            exact (Fin.cycleIcc i (Fin.last nA)).bijective,
          fun x => by
            refine Fin.addCases (fun a => ?_) (fun a => ?_) x
            · simp only [Function.comp_apply, Fin.append_left, Fin.succAbove_last,
                show (Fin.castSucc a : Fin (nA + 1)) = Fin.castAdd 1 a from rfl]
            · fin_cases a
              simp [Fin.append_right, Fin.succAbove_zero]⟩)
        t
```

lemma crossToSlot_round_trip_of_unit [source]

Round trip for `crossToSlot` at an arbitrary slot, with the colors in their composite form. The general-color statement `crossToSlot_raise_lower_round_trip` is obtained from this by substituting its three color equalities.

```
private lemma crossToSlot_round_trip_of_unit {nA : ℕ} {c : Fin (nA + 1) → C} {d : C}
    (i : Fin (nA + 1))
    (M₁ : Tensor S ![S.τ (c i), d]) (M₂ : Tensor S ![S.τ d, c i])
    (hM : crossToEnd (Fin.last 1) (0 : Fin 2) rfl M₁ M₂ =
      permT (id : Fin 2 → Fin 2) (IsReindexing.unitTensor_pair rfl)
        (unitTensor (S := S) (c i)))
    (t : Tensor S c) :
    crossToSlot (S := S) i (0 : Fin 2)
        (by rw [Function.update_self] : S.τ (Function.update c i d i) = S.τ d) M₂
        (crossToSlot (S := S) i (0 : Fin 2) rfl M₁ t) =
      permT (id : Fin (nA + 1) → Fin (nA + 1)) (IsReindexing.update_update_of_eq i rfl) t
```

lemma crossToSlot_raise_lower_round_trip [source]

Raising then lowering a named index, with every color threaded propositionally. For a rank-2 metric `M : ![a, d]` and an inverse `M' : ![b, e]` collapsing to the unit tensor (`M · M' = δ`), at any rank and any slot `i`: contracting slot `i` with `M`, then contracting

the result with `M'`, is the identity up to the `Function.update` color reindex.

No color is pinned to a composite. `ha : τ (c i) = a` places the metric's contracted slot, `hb : τ d = b` the inverse's, and `he : c i = e` the surviving one, so a caller whose metric and inverse sit at literal colors supplies them directly, with no transporting `permT` on either factor. Lowering then raising is this same statement with `M` and `M'` exchanged and the reversed collapse `M' · M = δ`; neither collapse follows from the other here.

```
lemma crossToSlot_raise_lower_round_trip {nA : ℕ} {c : Fin (nA + 1) → C} {a b d e : C}
    (i : Fin (nA + 1)) (he : c i = e) (ha : S.τ (c i) = a) (hb : S.τ d = b)
    (M : Tensor S ![a, d]) (M' : Tensor S ![b, e])
    (hM : crossToEnd (Fin.last 1) (0 : Fin 2) hb M M' =
      permT (id : Fin 2 → Fin 2)
        (IsReindexing.unitTensor_pair ((congrArg S.τ he.symm).trans ha))
        (unitTensor (S := S) e))
    (t : Tensor S c) :
    crossToSlot (S := S) i (0 : Fin 2)
        (by rw [Function.update_self]; exact hb : S.τ (Function.update c i d i) = b) M'
        (crossToSlot (S := S) i (0 : Fin 2) ha M t) =
      permT (id : Fin (nA + 1) → Fin (nA + 1)) (IsReindexing.update_update_of_eq i he) t
```

`lemma crossToSlotInv_crossToSlot` [source]

Lowering undoes raising: `crossToSlot_raise_lower_round_trip` with the color cast absorbed into `crossToSlotInv`.

```
lemma crossToSlotInv_crossToSlot (t : Tensor S c) :
    crossToSlotInv i he hb M' (crossToSlot i (0 : Fin 2) ha M t) = t
```

`lemma crossToSlot_crossToSlotInv` [source]

Raising undoes lowering. The reversed collapse `M' · M = δ` is the round trip at the color list `Function.update c i d` with the pair exchanged, which gives the lowering half a left inverse and hence injectivity; that upgrades `crossToSlotInv_crossToSlot` to a two-sided inverse.

```
lemma crossToSlot_crossToSlotInv (t : Tensor S (Function.update c i d)) :
    crossToSlot i (0 : Fin 2) ha M (crossToSlotInv i he hb M' t) = t
```

`def crossToSlotEquiv` [source]

Raising and lowering a named index against a mutually inverse pair, as a linear equivalence: the two color assignments `c` and `Function.update c i d` carry the same tensors. `M` and `M'` are matched only propositionally and `d` is unrelated to `c i`, so this covers a metric pairing two unrelated colors as well as a metric and its inverse at `S.τ (c i)`.

```
noncomputable def crossToSlotEquiv : Tensor S c ≃ₗ[k] Tensor S (Function.update c i d)
```

2.82 `Physlib.Relativity.Tensors.Dual`

`Physlib/Relativity/Tensors/Dual.lean` on GitHub.

def toDualMapAtIndex [source]

The linear map between `S.Tensor c` and `S.Tensor (Function.update c i (S.τ (c i)))` formed by contracting the index `i` with the metric tensor.

```
noncomputable def toDualMapAtIndex : {n : ℕ} → {c : Fin n → C} → (i : Fin n) →
    S.Tensor c →ₗ[k] S.Tensor (Function.update c i (S.τ (c i)))
  | 0, _, i => i.elim0
  | _ + 1, c, i => crossToSlot i (0 : Fin 2) rfl (metricTensor (S := S) (S.τ (c i)))
```

lemma crossToEnd_dual_metricTensor_metricTensor [source]

The metric tensor at `S.τ c` contracted with the metric tensor at `c` is the unit tensor at `c`.

```
lemma crossToEnd_dual_metricTensor_metricTensor {c : C} :
    crossToEnd (Fin.last 1) (0 : Fin 2) (S.τ_τ_apply c) (metricTensor (S := S) (S.τ c))
        (metricTensor (S := S) c) =
      permT (id : Fin 2 → Fin 2) (IsReindexing.unitTensor_pair rfl) (unitTensor (S := S) c)
```

lemma crossToEnd_metricTensor_metricTensor_eq_dual_unit [source]

The metric tensor at `c` contracted with the metric tensor at `S.τ c` is the unit tensor at `S.τ c`.

```
lemma crossToEnd_metricTensor_metricTensor_eq_dual_unit {c : C} :
    crossToEnd (Fin.last 1) (0 : Fin 2) rfl (metricTensor (S := S) c)
        (metricTensor (S := S) (S.τ c)) =
      permT (id : Fin 2 → Fin 2) (IsReindexing.unitTensor_pair (S.τ_τ_apply c))
        (unitTensor (S := S) (S.τ c))
```

lemma toDualMapAtIndex_toDualMapAtIndex [source]

Dualising the index `i` twice returns the original tensor, up to the reindexing of the colors.

```
lemma toDualMapAtIndex_toDualMapAtIndex {n : ℕ} {c : Fin n → C}
    (i : Fin n) (t : S.Tensor c) :
    toDualMapAtIndex (S := S) i (toDualMapAtIndex (S := S) i t) =
      permT (id : Fin n → Fin n)
        (IsReindexing.update_update_of_eq i (by simp [τ_τ_apply])) t
```

def fromDualMapAtIndex [source]

The linear map between `S.Tensor (Function.update c i (S.τ (c i)))` and `S.Tensor c` formed by contracting the index `i` with the metric tensor. It is the inverse of `toDualMapAtIndex`.

```
noncomputable def fromDualMapAtIndex : {n : ℕ} → {c : Fin n → C} → (i : Fin n) →
    S.Tensor (Function.update c i (S.τ (c i))) →ₗ[k] S.Tensor c
  | 0, _, i => i.elim0
  | _ + 1, c, i => crossToSlotInv i rfl (S.τ_τ_apply (c i)) (metricTensor (S := S) (c i))
```

lemma fromDualMapAtIndex_toDualMapAtIndex [source]

```
@[simp]
lemma fromDualMapAtIndex_toDualMapAtIndex {n : ℕ} {c : Fin n → C} (i : Fin n) (t : S.Tensor c)
    :
    fromDualMapAtIndex (S := S) i (toDualMapAtIndex (S := S) i t) = t
```

lemma toDualMapAtIndex_fromDualMapAtIndex [source]

```
@[simp]
lemma toDualMapAtIndex_fromDualMapAtIndex {n : ℕ} {c : Fin n → C} (i : Fin n)
    (t : S.Tensor (Function.update c i (S.τ (c i)))) :
    toDualMapAtIndex (S := S) i (fromDualMapAtIndex (S := S) i t) = t
```

lemma toDualMapAtIndex_equivariant [source]

Dualising the index `i` commutes with the action of `G`.

```
@[simp]
lemma toDualMapAtIndex_equivariant {n : ℕ} {c : Fin n → C} (i : Fin n) (g : G) (t : S.Tensor c)
     :
    toDualMapAtIndex (S := S) i (g • t) = g • toDualMapAtIndex (S := S) i t
```

def toDualAtIndex [source]

The linear equivalence between `S.Tensor c` and `S.Tensor (Function.update c i (S.τ (c i)))` formed by contracting the index `i` with the metric tensor.

```
noncomputable def toDualAtIndex : {n : ℕ} → {c : Fin n → C} → (i : Fin n) →
    S.Tensor c ≃ₗ[k] S.Tensor (Function.update c i (S.τ (c i)))
  | 0, _, i => i.elim0
  | _ + 1, c, i =>
    crossToSlotEquiv i rfl rfl (S.τ_τ_apply (c i)) (metricTensor (S := S) (S.τ (c i)))
      (metricTensor (S := S) (c i)) crossToEnd_dual_metricTensor_metricTensor
      crossToEnd_metricTensor_metricTensor_eq_dual_unit
```

lemma toDualAtIndex_apply [source]

```
@[simp]
lemma toDualAtIndex_apply {n : ℕ} {c : Fin n → C} (i : Fin n) (t : S.Tensor c) :
    toDualAtIndex (S := S) i t = toDualMapAtIndex (S := S) i t
```

lemma toDualAtIndex_symm_apply [source]

```
@[simp]
lemma toDualAtIndex_symm_apply {n : ℕ} {c : Fin n → C} (i : Fin n)
    (t : S.Tensor (Function.update c i (S.τ (c i)))) :
    (toDualAtIndex (S := S) i).symm t = fromDualMapAtIndex (S := S) i t
```

2.83 `Physlib.Relativity.Tensors.Elab`

`Physlib/Relativity/Tensors/Elab.lean` on GitHub.

def indexExprIsJiggle [source]

Bool which is true if an index is a jiggle `τ(i)`.

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

def indexRemoveTau [source]

Removes a `τ(·)` wrapper from an index, leaving any other index unchanged. Pure, so it can be used to compare indices up to `τ` (e.g. when counting contractions in `withoutContrEval`).

```
def indexRemoveTau (stx : TSyntax `indexExpr) : TSyntax `indexExpr
```

def getJigglePos [source]

Returns the positions of indices which are "jiggled", i.e., of the form `τ(μ)`, these are the indices which are to be raised or lowered.

```
def getJigglePos (ind : List (TSyntax `indexExpr)) : TermElabM (List ℕ)
```

abbrev TensorExpressionOperator [source]

A Tensor expression operator is a map which takes in a pair consisting of a list of indices and a term and outputs a list of indices and a term, after applying a certain operation.

```
abbrev TensorExpressionOperator
```

def create [source]

The creation of a tensor from a syntax tree.

```
def TensorExpressionOperator.create (stx : Syntax) :
    TermElabM (List (TSyntax `indexExpr) × Term)
```

def jiggle [source]

The tensor expression operator which rises or lowers the indices of a tensor based on indices with `τ`-syntax.

```
def TensorExpressionOperator.jiggle : TensorExpressionOperator
```

def eval [source]

The tensor expression operator which evaluates indices.

```
def TensorExpressionOperator.eval : TensorExpressionOperator
```

def contr [source]

The tensor expression operator which contracts indices.

```
def TensorExpressionOperator.contr : TensorExpressionOperator
```

def prod [source]

The tensor expression operator which takes the product of two tensors.

```
def TensorExpressionOperator.prod : List (TSyntax `indexExpr) × Term →
    List (TSyntax `indexExpr) × Term →
    TermElabM (List (TSyntax `indexExpr) × Term)
```

def neg [source]

The tensor expression operator which negates a tensor.

```
def TensorExpressionOperator.neg : TensorExpressionOperator
```

def smul [source]

The tensor expression operator which multiplies a tensor by a scalar.

```
def TensorExpressionOperator.smul (c : Term) : TensorExpressionOperator
```

def action [source]

The tensor expression operator which acts on a tensor by a group element.

```
def TensorExpressionOperator.action (c : Term) : TensorExpressionOperator
```

def colorsDefEq [source]

Whether `T1` and `T2` elaborate to tensors of definitionally equal colour (equal type). Used to decide whether an identity reindexing between them may be dropped: the bare, un-`permT`'d term is well typed exactly when the two colours agree by `rfl`. The check elaborates speculatively and restores the elaborator state afterwards.

```
def colorsDefEq (T1 T2 : Term) : TermElabM Bool
```

def permWrap [source]

Wraps `T` in the reindexing `permT ![lPerm] IsReindexing.auto`.

```
def permWrap (lPerm : List ℕ) (T : Term) : TermElabM Term
```

def add [source]

The tensor expression operator which adds two tensors.

```
def TensorExpressionOperator.add : List (TSyntax `indexExpr) × Term →
    List (TSyntax `indexExpr) × Term →
    TermElabM (List (TSyntax `indexExpr) × Term)
```

def equal [source]

The tensor expression operator which equates two tensors.

```
def TensorExpressionOperator.equal : List (TSyntax `indexExpr) × Term →
    List (TSyntax `indexExpr) × Term →
    TermElabM (List (TSyntax `indexExpr) × Term)
```

def stripToTensorSelf [source]

Removes redundant `Tensorial.toTensor` coercions from an elaborated tensor expression: every subterm `Tensorial.toTensor t` whose `Tensorial` instance is the canonical instance on `S.Tensor c` (`Tensorial.self`) is replaced by `t` itself, since there the coercion is the identity. Coercions coming from genuine tensorial instances (e.g. on Lorentz vectors) carry real information and are left untouched.

```
def stripToTensorSelf (e : Expr) : Expr
```

2.84 Physlib.Relativity.Tensors.Evaluation

Physlib/Relativity/Tensors/Evaluation.lean on GitHub.

lemma evalT_permT [source]

Commuting evaluation with permutations.

```
lemma evalT_permT {n m : ℕ} {c : Fin (n + 1) → C} {c' : Fin (m + 1) → C}
    {σ : Fin (n + 1) → Fin (m + 1)}
    (h : IsReindexing c' c  σ) (i : Fin (n + 1)) (x : basisIdx (c i)) (t : Tensor S c') :
    evalT i x (permT _ h t) = permT _ (h.succAbove i)
      (evalT (σ i) (basisIdxCongr (by simp [h.2]) x) t)
```

lemma contrT_evalT [source]

Commuting evaluation with contraction. Evaluating index `k` and then contracting the pair `i j` equals contracting the corresponding pair `k.succAbove i`, `k.succAbove j` and then evaluating the residual index, up to the identity reindexing `IsReindexing.succAbove_succSuccAbove_co`

```
lemma contrT_evalT {n : ℕ} {c : Fin (n + 1 + 1 + 1) → C}
    (k : Fin (n + 1 + 1 + 1)) (i j : Fin (n + 1 + 1)) (φ : basisIdx (c k))
    (hij : i ≠ j ∧ S.τ ((c ∘ k.succAbove) i) = (c ∘ k.succAbove) j) (t : Tensor S c) :
    contrT n i j hij (evalT k φ t) =
    permT id (.succAbove_succSuccAbove_comm k i j hij.1)
      (evalT (Fin.predPredAbove (k.succAbove i) (k.succAbove j) (by simp [hij.1]) k (by simp))
        (basisIdxCongr (by simp) φ)
        (contrT (n + 1) (k.succAbove i) (k.succAbove j) ⟨by simp [hij.1], hij.2⟩ t))
```

lemma evalT_prodT_right [source]

Evaluating an index in the right factor of a tensor product commutes with forming the product, up to the identity reindexing which identifies the two ways of removing that index from the appended color list.

```
lemma evalT_prodT_right {n n1 : ℕ} {c : Fin n → C} {c1 : Fin (n1 + 1) → C}
    (i : Fin (n1 + 1)) (x : basisIdx (c1 i)) (t : Tensor S c) (t1 : Tensor S c1) :
    prodT t (evalT i x t1) =
    permT id (IsReindexing.append_succAbove_natAdd (n := n) (n1 := n1) i)
      (evalT (Fin.natAdd (m := n1 + 1) n i) (basisIdxCongr (by simp) x) (prodT t t1))
```

lemma ext_of_evalT [source]

```
lemma ext_of_evalT {n : ℕ} {c : Fin (n + 1) → C} (t1 t2 : Tensor S c)
    (h : ∀ i φ, evalT i φ t1 = evalT i φ t2) :
    t1 = t2
```

lemma ext_of_evalT_index [source]

```
lemma ext_of_evalT_index {n : ℕ} {c : Fin (n + 1) → C} {t1 t2 : Tensor S c}
    (i : Fin (n + 1)) (h : ∀ φ, evalT i φ t1 = evalT i φ t2) :
    t1 = t2
```

2.85 Physlib.Relativity.Tensors.LeviCivita.Basic

Physlib/Relativity/Tensors/LeviCivita/Basic.lean on GitHub.

def euclidLeviCivita [source]

The Euclidean Levi-Civita symbol `ε_{ijkl}` in dimension 4.

```
def _root_.euclidLeviCivita (g : Fin 4 → Fin 4) : ℝ
```

lemma leviCivita_contract_three [source]

```
@[sorryful]
lemma leviCivita_contract_three : {ε4 | μ ν ρ σ ⊗ ε4 | τ(μ) τ(ν) τ(ρ) τ(τ) =
    (-6) • unitTensor (S := realLorentzTensor) Color.down | σ τ }^T
```

lemma leviCivita_contract_self [source]

```
@[sorryful, nolint checkType]
lemma leviCivita_contract_self :
    {ε4 | μ ν ρ σ ⊗ ε4 | τ(μ) τ(ν) τ(ρ) τ(σ)}^T.toField = - 24
```

2.86 Physlib.Relativity.Tensors.LeviCivita.Contractions

Physlib/Relativity/Tensors/LeviCivita/Contractions.lean on GitHub.

lemma euclidLeviCivita_symbol_contract_zero [source]

***Full Euclidean Levi-Civita contraction** `∑_b (ε4)_b · (ε4)_b = 24` at the symbol level: summing the square of every standard-basis component of `ε4` over all four `Fin 4` index slots, paired naively (no metric), counts the `4! = 24` permutations. The Lorentz contraction `ε^{μνρσ} ε_{μνρσ}` lowers one factor with `η` and equals `-24` instead.*

```
lemma euclidLeviCivita_symbol_contract_zero :
    ∑ g : (Fin 4 → Fin 4), euclidLeviCivita g * euclidLeviCivita g = 24
```

lemma euclidLeviCivita_symbol_contract_one [source]

***Triple Euclidean Levi-Civita contraction** `∑_h (ε4)_{a,h} · (ε4)_{b,h} = 6 · δ[a,b]` at the symbol level: contracting three of the four `Fin 4` component slots of `ε4` with the naive Kronecker pairing leaves one free pair `a, b` and the factor `3! = 6`. The Lorentz form carries an extra `det η = -1`.*

```
lemma euclidLeviCivita_symbol_contract_one (a b : Fin 4) :
    ∑ h : Fin 3 → Fin 4, euclidLeviCivita (Fin.cons a h) * euclidLeviCivita (Fin.cons b h)
      = 6 * ((kroneckerDelta a b : ℕ) : ℝ)
```

lemma euclidLeviCivita_symbol_contract_two [source]

***Double Euclidean Levi-Civita contraction** `∑_h (ε4)_{r,s,h} · (ε4)_{t,w,h} = 2 · (δ[r,t]·δ[s,w] - δ[r,w]·δ[s,t])` at the symbol level: contracting two of the four `Fin 4` component slots of `ε4` with the naive Kronecker pairing leaves two free pairs and the factor `2! = 2`. The Lorentz form carries an extra `det η = -1`.*

```
lemma euclidLeviCivita_symbol_contract_two (r s t w : Fin 4) :
    ∑ h : Fin 2 → Fin 4, euclidLeviCivita (Fin.cons r (Fin.cons s h))
          * euclidLeviCivita (Fin.cons t (Fin.cons w h))
      = 2 * (((kroneckerDelta r t : ℕ) : ℝ) * ((kroneckerDelta s w : ℕ) : ℝ)
          - ((kroneckerDelta r w : ℕ) : ℝ) * ((kroneckerDelta s t : ℕ) : ℝ))
```

2.87 Physlib.Relativity.Tensors.RealTensor.Representation.Contraction

Physlib/Relativity/Tensors/RealTensor/Representation/Contraction.lean on GitHub.

def Vector.contract [source]

The intertwining map defining the contraction of a contravariant Lorentz vector with a covariant Lorentz vector.

```
def Vector.contract : (Vector.rep.tprod CoVector.rep).IntertwiningMap
    (Representation.trivial ℝ (LorentzGroup d) ℝ) where
```

def CoVector.contract [source]

The intertwining map defining the contraction of a covariant Lorentz vector with a contravariant Lorentz vector.

```
def CoVector.contract : (CoVector.rep.tprod Vector.rep).IntertwiningMap
    (Representation.trivial ℝ (LorentzGroup d) ℝ) where
```

lemma Vector.contract_tmul [source]

```
lemma Vector.contract_tmul (φ : Vector d) (ψ : CoVector d) :
    Vector.contract (φ ⊗ₜ ψ) = ∑ i, φ i * ψ i
```

lemma CoVector.contract_tmul [source]

```
lemma CoVector.contract_tmul (φ : CoVector d) (ψ : Vector d) :
    CoVector.contract (φ ⊗ₜ ψ) = ∑ i, φ i * ψ i
```

lemma Vector.contract_basis_left [source]

```
lemma Vector.contract_basis_left (μ : Fin 1 ⊕ Fin d) (ψ : CoVector d) :
    Vector.contract (basis μ ⊗ₜ ψ) = ψ μ
```

lemma CoVector.contract_basis_left [source]

```
lemma CoVector.contract_basis_left (μ : Fin 1 ⊕ Fin d) (φ : Vector d) :
    CoVector.contract (basis μ ⊗ₜ φ) = φ μ
```

lemma Vector.contract_basis_right [source]

```
lemma Vector.contract_basis_right (φ : Vector d) (μ : Fin 1 ⊕ Fin d) :
    Vector.contract (φ ⊗ₜ basis μ) = φ μ
```

lemma CoVector.contract_basis_right [source]

```
lemma CoVector.contract_basis_right (ψ : CoVector d) (μ : Fin 1 ⊕ Fin d) :
    CoVector.contract (ψ ⊗ₜ basis μ) = ψ μ
```

lemma contract_eq_coVector_contract [source]

```
lemma Vector.contract_eq_coVector_contract (φ : Vector d) (ψ : CoVector d) :
    Vector.contract (φ ⊗ₜ ψ) = CoVector.contract (ψ ⊗ₜ φ)
```

lemma Vector.contract_rep [source]

```
lemma Vector.contract_rep (Λ : LorentzGroup d) (φ : Vector d) (ψ : CoVector d) :
    Vector.contract ((Vector.rep Λ φ) ⊗ₜ (CoVector.rep Λ ψ)) = Vector.contract (φ ⊗ₜ ψ)
```

lemma CoVector.contract_rep [source]

```
lemma CoVector.contract_rep (Λ : LorentzGroup d) (φ : CoVector d) (ψ : Vector d) :
    CoVector.contract ((CoVector.rep Λ φ) ⊗ₜ (Vector.rep Λ ψ)) = CoVector.contract (φ ⊗ₜ ψ)
```

2.88 Physlib.Relativity.Tensors.Reindexing

Physlib/Relativity/Tensors/Reindexing.lean on GitHub.

lemma append_castAdd_natAdd_eq_id [source]

Splitting `Fin (m + n)` into its two blocks and reassembling them is the identity.

```
lemma append_castAdd_natAdd_eq_id {m n : ℕ} :
    Fin.append (Fin.castAdd n) (Fin.natAdd m) = (id : Fin (m + n) → Fin (m + n))
```

lemma append_succAbove_const_eq_cycleIcc [source]

Moving slot `i` to the end is the cycle `[i, last]`: the block map that lists the `i.succAbove` survivors in order and then `i` is `Fin.cycleIcc i (Fin.last n)`.

```
lemma append_succAbove_const_eq_cycleIcc {n : ℕ} (i : Fin (n + 1)) :
    Fin.append i.succAbove (fun _ : Fin 1 => i) = ⇑(Fin.cycleIcc i (Fin.last n))
```

lemma swap [source]

```
lemma swap {n : ℕ} {c : Fin n → C} (i j : Fin n) :
    IsReindexing c (c ∘ Equiv.swap i j) (Equiv.swap i j)
```

lemma succAbove_of_succAbove_eq [source]

The conclusion of `succAbove` from an explicit survivor relabelling: if `σ'` fills the square `(σ i).succAbove ∘ σ' = σ ∘ i.succAbove`, carrying the complement of `i` to the complement of `σ i`, then it is a reindexing of the two shortened colour lists. The square forces `σ'` to be injective, hence bijective, so no bijectivity hypothesis is needed; this is why the statement is restricted to equal lengths.

Where `succAbove` builds the relabelling from `σ` as a `dite` composite, the map here is the caller's, which is what lets it stand in the statement of a lemma the caller instantiates.

```
lemma succAbove_of_succAbove_eq {n : ℕ} {c c1 : Fin (n + 1) → C}
    {σ : Fin (n + 1) → Fin (n + 1)} {σ' : Fin n → Fin n} (i : Fin (n + 1))
    (h : IsReindexing c c1 σ) (hσ' : (σ i).succAbove ∘ σ' = σ ∘ i.succAbove) :
    IsReindexing (c ∘ (σ i).succAbove) (c1 ∘ i.succAbove) σ'
```

lemma succSuccAbove_succAbove_comm [source]

```
lemma succSuccAbove_succAbove_comm {n : ℕ} {c : Fin (n + 1 + 1 + 1) → C}
    (k : Fin (n + 1)) (i j : Fin (n + 1 + 1 + 1)) :
```

lemma move_last [source]

Splitting a list of colours `c : Fin (n + 1) → C` at an arbitrary slot `i`, rather than at the last one as in `append_succ_last`: the block map that lists the `i.succAbove` survivors and then `i` itself matches `c` with the survivors of `i` followed by the surviving entry `c1 1` of a rank-two list `c1` whose second entry is slot `i`'s colour.

```
lemma move_last {n : ℕ} {c : Fin (n + 1) → C} {c1 : Fin 2 → C} (i : Fin (n + 1))
    (hc : c i = c1 1) :
    IsReindexing c
      (Fin.append (c ∘ i.succAbove) (c1 ∘ (0 : Fin 2).succAbove))
      (Fin.append i.succAbove (fun _ : Fin 1 => i))
```

lemma update_update_of_eq [source]

Updating slot `i` of `c` to `d` and then back to `e = c i` returns `c`, no other slot moving: the colour cast a round trip of two contractions at slot `i` generates.

```
lemma update_update_of_eq {n : ℕ} {c : Fin n → C} {d e : C} (i : Fin n) (he : c i = e) :
    IsReindexing c (Function.update (Function.update c i d) i e) (id : Fin n → Fin n)
```

2.89 Physlib.Relativity.Tensors.TensorSpecies.Basic

Physlib/Relativity/Tensors/TensorSpecies/Basic.lean on GitHub.

lemma basisIdxCongr_symm [source]

```
lemma basisIdxCongr_symm {c c1 : C} (h : c = c1) (i : basisIdx c1) :
    (basisIdxCongr h).symm i = basisIdxCongr h.symm i
```

2.90 Physlib.SpaceAndTime.Time.Derivatives

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

lemma deriv_add [source]

```
lemma deriv_add [NormedAddCommGroup M] [NormedSpace ℝ M] (f g : Time → M)
    (hf : DifferentiableAt ℝ f t) (hg : DifferentiableAt ℝ g t) :
    ∂ₜ (fun s => f s + g s) t = ∂ₜ f t + ∂ₜ g t
```

lemma deriv_fun_sum [source]

```
lemma deriv_fun_sum {ι : Type*} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (s : Finset ι) (a : ι → Time → M) (ha : ∀ i ∈ s, DifferentiableAt ℝ (a i) t) :
    ∂ₜ (fun x => ∑ i ∈ s, a i x) t = ∑ i ∈ s, ∂ₜ (a i) t
```

lemma deriv_mul_const [source]

```
lemma deriv_mul_const (f : Time → ℝ) (c : ℝ) (hf : DifferentiableAt ℝ f t) :
    ∂ₜ (fun s => f s * c) t = ∂ₜ f t * c
```

lemma val_contDiff [source]

```
@[fun_prop]
lemma val_contDiff {n : WithTop ℕ∞} : ContDiff ℝ n Time.val
```

lemma deriv_space [source]

```
lemma deriv_space {d : ℕ} {f : Time → Space d}
    (hf : Differentiable ℝ f) (t : Time) (i : Fin d) :
    deriv (fun s => f s i) t = deriv f t i
```

2.91 Physlib.SpaceAndTime.Time.MatrixDerivatives

Physlib/SpaceAndTime/Time/MatrixDerivatives.lean on GitHub.

lemma matrix_transpose [source]

The transpose of a differentiable matrix-valued function is differentiable (cf. `Continuous.matrix_transpose`).

```
lemma DifferentiableAt.matrix_transpose {E : Type*} [NormedAddCommGroup E] [NormedSpace ℝ E]
    {A : E → Matrix (Fin d) (Fin d) ℝ} {t : E} (hA : DifferentiableAt ℝ A t) :
    DifferentiableAt ℝ (fun s => (A s)ᵀ) t
```

lemma deriv_matrix_mul [source]

Product rule for the time derivative of a product of matrix-valued functions.

```
lemma deriv_matrix_mul (A B : Time → Matrix (Fin d) (Fin d) ℝ) (t : Time)
    (hA : DifferentiableAt ℝ A t) (hB : DifferentiableAt ℝ B t) :
    ∂ₜ (fun s => A s * B s) t = A t * ∂ₜ B t + ∂ₜ A t * B t
```

lemma deriv_matrix_transpose [source]

The time derivative commutes with transpose.

```
lemma deriv_matrix_transpose (A : Time → Matrix (Fin d) (Fin d) ℝ) (t : Time)
    (hA : DifferentiableAt ℝ A t) :
    ∂ₜ (fun s => (A s)ᵀ) t = (∂ₜ A t)ᵀ
```

lemma deriv_matrix_apply [source]

The time derivative commutes with taking a matrix entry.

```
lemma deriv_matrix_apply (A : Time → Matrix (Fin d) (Fin d) ℝ) (t : Time)
    (hA : DifferentiableAt ℝ A t) (i j : Fin d) :
    ∂ₜ (fun s => A s i j) t = (∂ₜ A t) i j
```

2.92 Physlib.Units.Dimension

Physlib/Units/Dimension.lean on GitHub.

instance Mul (Dimension B) [source]

```
instance : Mul (Dimension B) where
```

lemma mul_exponent [source]

```
@[simp]
lemma mul_exponent (d1 d2 : Dimension B) (b : B) :
    (d1 * d2).exponent b = d1.exponent b + d2.exponent b
```

instance One (Dimension B) [source]

```
instance : One (Dimension B) where
```

lemma one_exponent [source]

```
@[simp]
lemma one_exponent (b : B) : (1 : Dimension B).exponent b = 0
```

instance CommGroup (Dimension B) [source]

```
instance : CommGroup (Dimension B) where
```

lemma inv_exponent [source]

```
@[simp]
lemma inv_exponent (d : Dimension B) (b : B) : d⁻¹.exponent b = -d.exponent b
```

lemma div_exponent [source]

```
@[simp]
lemma div_exponent (d1 d2 : Dimension B) (b : B) :
    (d1 / d2).exponent b = d1.exponent b - d2.exponent b
```

lemma npow_exponent [source]

```
@[simp]
lemma npow_exponent (d : Dimension B) (n : ℕ) (b : B) :
    (d ^ n).exponent b = n • d.exponent b
```

instance Pow (Dimension B) ℚ [source]

```
instance : Pow (Dimension B) ℚ where
```

lemma qpow_exponent [source]

```
@[simp]
lemma qpow_exponent (d : Dimension B) (q : ℚ) (b : B) :
    (d ^ q).exponent b = d.exponent b * q
```

instance DecidableEq (Dimension B) [source]

Decidable equality of dimensions over a finite basis `B`.

```
instance [Fintype B] : DecidableEq (Dimension B)
```

def single [source]

The base-dimension vector for `b : B`: exponent `1` at `b`, `0` elsewhere. This is the generic analogue of the named generators `L`, `T`, ...

```
def single [DecidableEq B] (b : B) : Dimension B
```

lemma single_exponent [source]

```
@[simp]
lemma single_exponent [DecidableEq B] (b b' : B) :
    (single b).exponent b' = if b' = b then 1 else 0
```

def extend [source]

Change of basis along a map `f : B → B'` of base dimensions: reindex a dimension over `B` into one over `B'` by placing each exponent at its image. For an embedding `f` (injective) this preserves every exponent (`extend_exponent_apply`), so a dimension in one system re-expresses faithfully in an extending one.

```
def extend {B' : Type} [Fintype B] [DecidableEq B'] (f : B → B') (d : Dimension B) :
    Dimension B'
```

lemma extend_exponent_apply [source]

```
@[simp]
lemma extend_exponent_apply {B' : Type} [Fintype B] [DecidableEq B'] {f : B → B'}
    (hf : Function.Injective f) (d : Dimension B) (b : B) :
    (extend f d).exponent (f b) = d.exponent b
```

def extendHom [source]

`extend f` packaged as a monoid homomorphism of dimensions.

```
def extendHom {B' : Type} [Fintype B] [DecidableEq B'] (f : B → B') :
    Dimension B →* Dimension B' where
```

structure Embedding [source]

*A **dimension embedding** `B ↪ B'`: an injective monoid homomorphism of dimensions. As a `MonoidHom` it is dimension-preserving (it respects products, inverses and rational powers); injectivity makes it a faithful inclusion of the basis `B` into `B'`. Cross-basis dimension injections are produced as `Embedding`s so that dimension-preservation holds by construction.*

```
structure Embedding (B B' : Type) where
  toHom : Dimension B →* Dimension B'
  inj : Function.Injective toHom
```

structure Projection [source]

*A **dimension projection** `B' ↠ B`: a surjective monoid homomorphism of dimensions. As a `MonoidHom` it is truth-preserving, but it is lossy — it reduces a richer basis `B'` onto a coarser basis `B`, collapsing the base dimensions that `B` does not track.*

```
structure Projection (B' B : Type) where
  toHom : Dimension B' →* Dimension B
  surj : Function.Surjective toHom
```

def ofBasis [source]

*An injective *basis* map `f : B → B'` induces a dimension embedding, via `extend`. This is the label-level case: it sends each base dimension of `B` to a base dimension of `B'`, so it is automatically dimension-preserving and faithful.*

```
def Embedding.ofBasis {B B' : Type} [Fintype B] [DecidableEq B']
    (f : B → B') (hf : Function.Injective f) : Embedding B B' where
```

2.93 Physlib.Units.ISQBridge

Physlib/Units/ISQBridge.lean on GitHub.

def toISQFun [source]

The function underlying `toISQHom`: fix length, mass, time and temperature, and send PhysLib's charge generator to the derived ISQ charge `I · T` (the current exponent is the charge exponent, and the time exponent absorbs it).

```
def toISQFun (d : Dimension LTMCTDimensionBase) : Dimension ISQDimensionBase
```

def toISQHom [source]

The dimension-preserving embedding of PhysLib dimensions into the ISQ dimensions.

```
def toISQHom : Dimension LTMCTDimensionBase →* Dimension ISQDimensionBase where
```

lemma toISQHom_apply [source]

`toISQHom` applied to a dimension is `toISQFun`.

```
lemma toISQHom_apply (d : Dimension LTMCTDimensionBase) : toISQHom d = toISQFun d
```

def fromISQFun [source]

The function underlying `fromISQHom`: fix length, mass and temperature, read electric current as charge/time (the charge exponent is the current exponent, and the time exponent subtracts it), and drop amount of substance and luminous intensity.

```
def fromISQFun (d : Dimension ISQDimensionBase) : Dimension LTMCTDimensionBase
```

def fromISQHom [source]

The truth-preserving reduction of ISQ dimensions onto PhysLib's.

```
def fromISQHom : Dimension ISQDimensionBase →* Dimension LTMCTDimensionBase where
```

lemma fromISQHom_apply [source]

`fromISQHom` applied to a dimension is `fromISQFun`.

```
lemma fromISQHom_apply (d : Dimension ISQDimensionBase) : fromISQHom d = fromISQFun d
```

lemma fromISQHom_comp_toISQHom [source]

The projection recovers PhysLib's basis from its image under the embedding: `fromISQHom ∘ toISQHom = id`.

```
lemma fromISQHom_comp_toISQHom :
    fromISQHom.comp toISQHom = MonoidHom.id (Dimension LTMCTDimensionBase)
```

lemma toISQHom_injective [source]

`toISQHom` is injective: PhysLib dimensions include faithfully into ISQ.

```
lemma toISQHom_injective : Function.Injective toISQHom
```

lemma fromISQHom_surjective [source]

`fromISQHom` is surjective: every PhysLib dimension is the reduction of some ISQ dimension (namely its own embedding).

```
lemma fromISQHom_surjective : Function.Surjective fromISQHom
```

def ltmctToISQ [source]

PhysLib's dimensions embed into the ISQ dimensions (charge ↦ `I · T`).

```
def ltmctToISQ : Embedding LTMCTDimensionBase ISQDimensionBase where
```

def isqToLTMCT [source]

The ISQ dimensions project onto PhysLib's, reading current as charge/time and forgetting amount of substance and luminous intensity.

```
def isqToLTMCT : Projection ISQDimensionBase LTMCTDimensionBase where
```

lemma isqToLTMCT_comp_ltmctToISQ [source]

The projection is a retraction of the embedding: `isqToLTMCT ∘ ltmctToISQ = id`. So PhysLib's basis embeds faithfully into ISQ and is recovered by the projection, but not conversely — amount of substance and luminous intensity cannot be recovered.

```
lemma isqToLTMCT_comp_ltmctToISQ :
    isqToLTMCT.toHom.comp ltmctToISQ.toHom = MonoidHom.id (Dimension LTMCTDimensionBase)
```

lemma toISQHom_C⌈ [source]

*The embedding is physically faithful on charge: PhysLib's charge generator `C⌈` maps to the *derived* ISQ charge `I · T`.*

```
lemma toISQHom_C⌈ : toISQHom C⌈ = ISQDimensionBase.charge
```

2.94 Physlib.Units.ISQDimensionBase

Physlib/Units/ISQDimensionBase.lean on GitHub.

inductive ISQDimensionBase [source]

The seven ISQ base quantities (ISO/IEC 80000-1; VIM 3rd edition). Unlike `LTMCTDimensionBase`, electric current is the electromagnetic base quantity (so charge is derived), and amount of substance and luminous intensity are further base quantities.

```
inductive ISQDimensionBase where
  | length
  | mass
  | time
  | current
  | temperature
  | amount
  | luminousIntensity
```

lemma card_eq_seven [source]

The ISQ has seven base quantities.

```
lemma card_eq_seven : Fintype.card ISQDimensionBase = 7
```

def charge [source]

*Electric charge is a *derived* dimension in the ISQ: `Q = I · T`.*

```
def charge : Dimension ISQDimensionBase
```

lemma charge_exponent_current [source]

The derived charge dimension has electric-current exponent `1`.

```
lemma charge_exponent_current : charge.exponent .current = 1
```

lemma charge_exponent_time [source]

The derived charge dimension has time exponent `1`.

```
lemma charge_exponent_time : charge.exponent .time = 1
```

lemma charge_exponent_mass [source]

The derived charge dimension has mass exponent `0`.

```
lemma charge_exponent_mass : charge.exponent .mass = 0
```

lemma single_amount_ne_one [source]

Amount of substance is an independent base dimension (absent from `LTMCTDimensionBase`).

```
lemma single_amount_ne_one : single (.amount : ISQDimensionBase) ≠ 1
```

lemma single_luminousIntensity_ne_one [source]

Luminous intensity is an independent base dimension (absent from `LTMCTDimensionBase`).

```
lemma single_luminousIntensity_ne_one :
    single (.luminousIntensity : ISQDimensionBase) ≠ 1
```

2.95 Physlib.Units.LTMCTDimensionBase

Physlib/Units/LTMCTDimensionBase.lean on GitHub.

inductive LTMCTDimensionBase [source]

*PhysLib's default basis of base dimensions — `length`, `time`, `mass`, `charge`, `temperature`. Note this is *charge*-based, so it is not the SI/ISQ base-quantity set; see `Dimension`.*

```
inductive LTMCTDimensionBase where
  | length
  | time
```

```
  | mass
  | charge
  | temperature
```

def length [source]

The length exponent of a `LTMCTDimensionBase` dimension.

```
def length (d : Dimension LTMCTDimensionBase) : ℚ
```

def time [source]

The time exponent of a `LTMCTDimensionBase` dimension.

```
def time (d : Dimension LTMCTDimensionBase) : ℚ
```

def mass [source]

The mass exponent of a `LTMCTDimensionBase` dimension.

```
def mass (d : Dimension LTMCTDimensionBase) : ℚ
```

def charge [source]

The charge exponent of a `LTMCTDimensionBase` dimension.

```
def charge (d : Dimension LTMCTDimensionBase) : ℚ
```

def temperature [source]

The temperature exponent of a `LTMCTDimensionBase` dimension.

```
def temperature (d : Dimension LTMCTDimensionBase) : ℚ
```

def ofLTMCTDimensionBase [source]

Build a `LTMCTDimensionBase` dimension from its five exponents, in the order `(length, time, mass, charge, temperature)`.

```
def ofLTMCTDimensionBase (length time mass charge temperature : ℚ) : Dimension
    LTMCTDimensionBase
```

lemma ofLTMCTDimensionBase_length [source]

```
@[simp]
lemma ofLTMCTDimensionBase_length (l t m c θ : ℚ) :
    (ofLTMCTDimensionBase l t m c θ).length = l
```

lemma ofLTMCTDimensionBase_time [source]

```
@[simp]
lemma ofLTMCTDimensionBase_time (l t m c θ : ℚ) :
    (ofLTMCTDimensionBase l t m c θ).time = t
```

lemma ofLTMCTDimensionBase_mass [source]

```
@[simp]
lemma ofLTMCTDimensionBase_mass (l t m c θ : ℚ) :
    (ofLTMCTDimensionBase l t m c θ).mass = m
```

lemma ofLTMCTDimensionBase_charge [source]

```
@[simp]
lemma ofLTMCTDimensionBase_charge (l t m c θ : ℚ) :
    (ofLTMCTDimensionBase l t m c θ).charge = c
```

lemma ofLTMCTDimensionBase_temperature [source]

```
@[simp]
lemma ofLTMCTDimensionBase_temperature (l t m c θ : ℚ) :
    (ofLTMCTDimensionBase l t m c θ).temperature = θ
```

lemma L⌈_eq_single [source]

```
lemma L⌈_eq_single : L⌈ = single .length
```

lemma T⌈_eq_single [source]

```
lemma T⌈_eq_single : T⌈ = single .time
```

lemma M⌈_eq_single [source]

```
lemma M⌈_eq_single : M⌈ = single .mass
```

lemma C⌈_eq_single [source]

```
lemma C⌈_eq_single : C⌈ = single .charge
```

lemma 0⌈_eq_single [source]

```
lemma 0⌈_eq_single : 0⌈ = single .temperature
```

2.96 Physlib.Units.ParametricDimensionExamples

Physlib/Units/ParametricDimensionExamples.lean on GitHub.

inductive Info [source]

A basis of information-theoretic base dimensions.

```
inductive Info
  | bit
  | symbol
```

def bitDim [source]

The `bit` base dimension.

```
def bitDim : Dimension Info
```

def symbolDim [source]

The `symbol` base dimension.

```
def symbolDim : Dimension Info
```

2.97 Physlib.Units.ParametricUnits

Physlib/Units/ParametricUnits.lean on GitHub.

structure UnitScale [source]

A choice of unit for each base dimension of `B`: a positive-real magnitude per base. This is the basis-generic form of `UnitChoices`.

```
structure UnitScale (B : Type) where
  scale : B → ℝ≥0
  scale_pos : ∀ b, 0 < scale b
```

lemma ratio_ne_zero [source]

```
lemma ratio_ne_zero (u1 u2 : UnitScale B) (b : B) : u1.scale b / u2.scale b ≠ 0
```

def dimScale [source]

The dimension-scaling homomorphism, generic in the basis `B`: a quantity of dimension `d` rescales by `∏ b, (u1 b / u2 b) ^ d.exponent b` when changing the unit choice from `u1` to `u2`. This is the basis-generic form of `UnitChoices.dimScale`.

```
noncomputable def dimScale [Fintype B] (u1 u2 : UnitScale B) : Dimension B →* ℝ≥0 where
```

lemma dimScale_self [source]

```
@[simp]
lemma dimScale_self [Fintype B] (u : UnitScale B) (d : Dimension B) :
    dimScale u u d = 1
```

lemma dimScale_one [source]

```
@[simp]
lemma dimScale_one [Fintype B] (u1 u2 : UnitScale B) :
    dimScale u1 u2 1 = 1
```

lemma dimScale_transitive [source]

The scaling is transitive (a cocycle in the unit choices).

```
lemma dimScale_transitive [Fintype B] (u1 u2 u3 : UnitScale B) (d : Dimension B) :
    dimScale u1 u2 d * dimScale u2 u3 d = dimScale u1 u3 d
```

def toScale [source]

Read a five-field `UnitChoices` as a `UnitScale` over `LTMCTDimensionBase`.

```
noncomputable def toScale (u : UnitChoices) : UnitScale LTMCTDimensionBase where
```

2.98 Physlib.Units.UnitDependent

Physlib/Units/UnitDependent.lean on GitHub.

instance MulAction ℝ≥0 (DimSet M d) [source]

```
instance (M : Type) [MulAction ℝ≥0 M] [MulUnitDependent M] (d : Dimension LTMCTDimensionBase) :
    MulAction ℝ≥0 (DimSet M d) where
```

instance CarriesDimension (DimSet M d) [source]

```
instance (M : Type) [MulAction ℝ≥0 M] [MulUnitDependent M] (d : Dimension LTMCTDimensionBase) :
    CarriesDimension (DimSet M d) where
```

2.99 Physlib.Units.WithDim.Basic

Physlib/Units/WithDim/Basic.lean on GitHub.

instance HasDim (WithDim d M) [source]

```
instance (d : Dimension LTMCTDimensionBase) (M : Type) : HasDim (WithDim d M) where
```

instance Inhabited (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [Inhabited M] : Inhabited (WithDim d M) where
```

instance Zero (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [Zero M] : Zero (WithDim d M) where
```

instance Add (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [Add M] : Add (WithDim d M) where
```

instance Neg (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [Neg M] : Neg (WithDim d M) where
```

instance Sub (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [Sub M] : Sub (WithDim d M) where
```

instance AddSemigroup (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [AddSemigroup M] :
    AddSemigroup (WithDim d M) where
```

instance AddCommSemigroup (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [AddCommSemigroup M] :
    AddCommSemigroup (WithDim d M) where
```

instance AddMonoid (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [AddMonoid M] :
    AddMonoid (WithDim d M) where
```

instance AddCommMonoid (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [AddCommMonoid M] :
    AddCommMonoid (WithDim d M) where
```

instance AddGroup (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [AddGroup M] :
    AddGroup (WithDim d M) where
```

instance AddCommGroup (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [AddCommGroup M] :
    AddCommGroup (WithDim d M) where
```

instance LE (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [LE M] : LE (WithDim d M) where
```

instance LT (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [LT M] : LT (WithDim d M) where
```

instance Preorder (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [Preorder M] :
    Preorder (WithDim d M) where
```

instance PartialOrder (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [PartialOrder M] :
    PartialOrder (WithDim d M) where
```

instance MulAction ℝ≥0 (WithDim d M) [source]

```
instance {B : Type} (d : Dimension B) (M : Type) [MulAction ℝ≥0 M] :
    MulAction ℝ≥0 (WithDim d M) where
```

instance HMul (WithDim d1 ℝ) (WithDim d2 ℝ) (WithDim (d1 * d2) ℝ) [source]

```
instance {B : Type} {d1 d2 : Dimension B} :
    HMul (WithDim d1 ℝ) (WithDim d2 ℝ) (WithDim (d1 * d2) ℝ) where
```

instance DMul (WithDim d1 ℝ) (WithDim d2 ℝ) (WithDim (d1 * d2) ℝ) [source]

```
instance {d1 d2 : Dimension LTMCTDimensionBase} :
    DMul (WithDim d1 ℝ) (WithDim d2 ℝ) (WithDim (d1 * d2) ℝ) where
```

instance HDiv (WithDim d1 ℝ) (WithDim d2 ℝ) (WithDim (d1 * d2⁻¹) ℝ) [source]

```
noncomputable instance {B : Type} (d1 d2 : Dimension B) :
    HDiv (WithDim d1 ℝ) (WithDim d2 ℝ) (WithDim (d1 * d2⁻¹) ℝ) where
```

2.100 PhyslibAlpha.ClassicalFieldTheory.Local.EulerLagrangeEquation

PhyslibAlpha/ClassicalFieldTheory/Local/EulerLagrangeEquation.lean on GitHub.

def SatisfiesEulerLagrange [source]

A field satisfies the local Euler-Lagrange equations for a local lagrangian when the Euler-Lagrange operator vanishes along that field.

```
def SatisfiesEulerLagrange (L : Lagrangian d m k)
    (f : Space d → EuclideanSpace ℝ (Fin m)) : Prop
```

lemma satisfiesEulerLagrange_iff [source]

```
@[simp]
lemma satisfiesEulerLagrange_iff (L : Lagrangian d m k)
    (f : Space d → EuclideanSpace ℝ (Fin m)) :
    SatisfiesEulerLagrange L f ↔ eulerLagrangeOp L f = 0
```

lemma eulerLagrangeOp_eq_zero [source]

```
lemma SatisfiesEulerLagrange.eulerLagrangeOp_eq_zero
    {L : Lagrangian d m k} {f : Space d → EuclideanSpace ℝ (Fin m)}
    (h : SatisfiesEulerLagrange L f) :
    eulerLagrangeOp L f = 0
```

lemma of_eulerLagrangeOp_eq_zero [source]

```
lemma SatisfiesEulerLagrange.of_eulerLagrangeOp_eq_zero
    {L : Lagrangian d m k} {f : Space d → EuclideanSpace ℝ (Fin m)}
    (h : eulerLagrangeOp L f = 0) :
    SatisfiesEulerLagrange L f
```

theorem isCritical_iff_satisfiesEulerLagrange_of_contDiff_and_smoothInCoordinates [source]

```
theorem isCritical_iff_satisfiesEulerLagrange_of_contDiff_and_smoothInCoordinates
    (L : Lagrangian d m k) (f : Space d → EuclideanSpace ℝ (Fin m)) (hf : ContDiff ℝ ∞ f)
    (hL : Lagrangian.SmoothInCoordinates L)
    (hbase : HasFiniteAction L f) :
    IsCritical L f ↔ SatisfiesEulerLagrange L f
```

theorem isCritical_iff_satisfiesEulerLagrange_of_admissibleForAction_and_smoothInCoordinates [source]

```
theorem isCritical_iff_satisfiesEulerLagrange_of_admissibleForAction_and_smoothInCoordinates
    (L : Lagrangian d m k) (f : Space d → EuclideanSpace ℝ (Fin m))
    (hLf : IsAdmissibleForAction L f)
    (hL : Lagrangian.SmoothInCoordinates L) :
    IsCritical L f ↔ SatisfiesEulerLagrange L f
```

2.101 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.ChargeBalance

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/ChargeBalance.lean on GitHub.

lemma coeff_aeval_diag [source]

Rescaling each variable `Xᵢ` by a constant `d i` multiplies the coefficient of the monomial `m` by `∏ᵢ (d i) ^ (m i)`.

```
lemma coeff_aeval_diag {σ R : Type*} [CommRing R] (d : σ → R) (f : MvPolynomial σ R)
    (m : σ →₀ ℕ) :
    coeff m (aeval (fun i => C (d i) * X i) f) = (m.prod fun i k => d i ^ k) * coeff m f
```

lemma coeff_eq_zero_of_charge_ne_zero [source]

***Charge balancing.** If each variable `Xᵢ` carries an integer charge `w i`, `c` is a phase of infinite order, and the polynomial `f` is invariant under the charge rotation `Xᵢ ↦ c^{wᵢ} Xᵢ`, then every monomial with nonzero net charge has vanishing coefficient.*

```
lemma coeff_eq_zero_of_charge_ne_zero {σ K : Type*} [Field K] (w : σ → ℤ) (c : K) (hc : c ≠ 0)
    (hroot : ∀ n : ℤ, c ^ n = 1 → n = 0) {f : MvPolynomial σ K}
    (hf : aeval (fun i => C (c ^ (w i)) * X i) f = f) {m : σ →₀ ℕ}
    (hm : ∑ i ∈ m.support, (m i : ℤ) * w i ≠ 0) :
    coeff m f = 0
```

2.102 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.EffectivePotential

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/EffectivePotential.lean on GitHub.

abbrev EffectivePotential [source]

A general potential of the Higgs field.

```
abbrev EffectivePotential : Type
```

def IsInvariant [source]

The proposition that the general potential is invariant under the global action of the gauge group.

```
def IsInvariant (V : EffectivePotential) : Prop
```

def HasMaxMassDimLE [source]

The proposition that the potential `V` has a maximum mass dimension less then or equal to `n` - also implying it is a polynomial.

```
def HasMaxMassDimLE (V : EffectivePotential) (n : ℕ) : Prop
```

lemma exists_comp_linear_poly [source]

A polynomial potential, restricted along any real-linear parametrisation `L` of field configurations, is a genuine polynomial in the parameters. This is the bookkeeping that lets the potential be evaluated on the field components of a gauge slice.

```
lemma HasMaxMassDimLE.exists_comp_linear_poly {V : EffectivePotential} {n : ℕ}
    (h : HasMaxMassDimLE V n) {ι : Type*} [Fintype ι] [DecidableEq ι]
    (L : (ι → ℝ) →ₗ[ℝ] TwoHiggsDoublet) :
    ∃ P : MvPolynomial ι ℝ, ∀ a : ι → ℝ, V (L a) = P.eval a
```

2.103 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.GaugeSlice

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/GaugeSlice.lean on GitHub.

def sliceHiggs [source]

The upper-triangular slice configuration `((z, 0), (w₀, w₁))`. It specialises to `repHiggs` when the components take their real "canonical frame" values.

```
def sliceHiggs (z w0 w1 : ℂ) : TwoHiggsDoublet where
```

def sliceR [source]

The slice as a real-linear map from the six real field parameters `(Re Φ1₀, Im Φ1₀, Re Φ2₀, Im Φ2₀, Re Φ2₁, Im Φ2₁)`.

```
def sliceR : (Fin 6 → ℝ) →ₗ[ℝ] TwoHiggsDoublet where
```

lemma repHiggs_eq_sliceHiggs [source]

The representative family is the real slice.

```
lemma repHiggs_eq_sliceHiggs (X : Fin 4 → ℝ) :
    repHiggs X = sliceHiggs (X 0) ((X 1 : ℂ) + Complex.I * (X 2 : ℂ)) (X 3)
```

lemma gaugeCartan_smul_sliceHiggs [source]

Hypercharge action of the Cartan phase on the slice: it multiplies the first components by `a` and the perpendicular second component by `ā`.

```
lemma gaugeCartan_smul_sliceHiggs (a : unitary ℂ) (z w0 w1 : ℂ) :
    GaugeGroupI.gaugeCartan a • sliceHiggs z w0 w1
      = sliceHiggs ((a : ℂ) * z) ((a : ℂ) * w0) ((star a : ℂ) * w1)
```

lemma ofU1Subgroup_smul_sliceHiggs [source]

Hypercharge action of the residual `U(1)` on the slice: it multiplies the perpendicular second component by `c⁶` and leaves the first components fixed.

```
lemma ofU1Subgroup_smul_sliceHiggs (c : unitary ℂ) (z w0 w1 : ℂ) :
    GaugeGroupI.ofU1Subgroup c • sliceHiggs z w0 w1
      = sliceHiggs z w0 ((c : ℂ) ^ 6 * w1)
```

def cartanRotParam [source]

The Cartan hypercharge phase `u`, transported to a rotation of the six real field parameters: it phases the first-component pairs by `u` and the perpendicular pair by `ū`.

```
def cartanRotParam (u : unitary ℂ) (a : Fin 6 → ℝ) : Fin 6 → ℝ
```

lemma gaugeCartan_smul_sliceR [source]

Acting by the Cartan phase on a slice configuration is the same as rotating its parameters.

```
lemma gaugeCartan_smul_sliceR (u : unitary ℂ) (a : Fin 6 → ℝ) :
    GaugeGroupI.gaugeCartan u • sliceR a = sliceR (cartanRotParam u a)
```

def resRotParam [source]

The residual `U(1)` phase `c`, transported to a rotation of the perpendicular parameter pair.

```
def resRotParam (c : unitary ℂ) (a : Fin 6 → ℝ) : Fin 6 → ℝ
```

lemma ofU1Subgroup_smul_sliceR [source]

Acting by the residual `U(1)` on a slice configuration rotates only the perpendicular pair.

```
lemma ofU1Subgroup_smul_sliceR (c : unitary ℂ) (a : Fin 6 → ℝ) :
    GaugeGroupI.ofU1Subgroup c • sliceR a = sliceR (resRotParam c a)
```

2.104 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.GaugeTorus

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/GaugeTorus.lean on GitHub.

def gaugeCartan [source]

The Cartan `SU(2)` gauge element `diag(a, ā)`, for `a` a phase.

```
noncomputable def gaugeCartan (a : unitary ℂ) : GaugeGroupI
```

lemma gaugeCartan_toU1 [source]

```
@[simp]
lemma gaugeCartan_toU1 (a : unitary ℂ) : (gaugeCartan a).toU1 = 1
```

lemma gaugeCartan_toSU2_coe [source]

```
@[simp]
lemma gaugeCartan_toSU2_coe (a : unitary ℂ) :
    ((gaugeCartan a).toSU2 : Matrix (Fin 2) (Fin 2) ℂ) = !![(a : ℂ), 0; 0, (star a : ℂ)]
```

lemma gaugeCartan_smul_eq [source]

The Cartan element acts as the diagonal matrix `diag(a, ā)`.

```
lemma gaugeCartan_smul_eq (a : unitary ℂ) (φ : HiggsVec) :
    gaugeCartan a • φ = WithLp.toLp 2 (!![(a : ℂ), 0; 0, (star a : ℂ)] *ᵥ φ.ofLp)
```

2.105 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.Invariants

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/Invariants.lean on GitHub.

lemma eval_aeval_comp [source]

Pushing an evaluation through an `aeval` substitution.

```
lemma eval_aeval_comp {R : Type*} [CommRing R] {κ ι : Type*} (x : ι → R)
    (f : κ → MvPolynomial ι R) (G : MvPolynomial κ R) :
    eval x (aeval f G) = eval (fun i => eval x (f i)) G
```

def cartanSubst [source]

The Cartan hypercharge rotation of the slice parameters, as a substitution of the polynomial variables.

```
noncomputable def cartanSubst (u : unitary ℂ) : Fin 6 → MvPolynomial (Fin 6) ℝ
```

lemma eval_cartanSubst [source]

```
lemma eval_cartanSubst (u : unitary ℂ) (a : Fin 6 → ℝ) :
    (fun k => MvPolynomial.eval a (cartanSubst u k)) = cartanRotParam u a
```

lemma aeval_cartanSubst_eq [source]

Gauge (Cartan) invariance of the potential forces the slice polynomial to be invariant under the hypercharge rotation of its variables.

```
lemma aeval_cartanSubst_eq {V : EffectivePotential} (hI : IsInvariant V)
    {P : MvPolynomial (Fin 6) ℝ} (hP : ∀ a, V (sliceR a) = P.eval a) (u : unitary ℂ) :
    aeval (cartanSubst u) P = P
```

def residualSubst [source]

The residual `U(1)` rotation of the perpendicular parameter, as a substitution.

```
noncomputable def residualSubst (c : unitary ℂ) : Fin 6 → MvPolynomial (Fin 6) ℝ
```

lemma eval_residualSubst [source]

```
lemma eval_residualSubst (c : unitary ℂ) (a : Fin 6 → ℝ) :
    (fun k => MvPolynomial.eval a (residualSubst c k)) = resRotParam c a
```

lemma aeval_residualSubst_eq [source]

Gauge (residual `U(1)`) invariance forces the slice polynomial to be invariant under the perpendicular rotation of its variables.

```
lemma aeval_residualSubst_eq {V : EffectivePotential} (hI : IsInvariant V)
    {P : MvPolynomial (Fin 6) ℝ} (hP : ∀ a, V (sliceR a) = P.eval a) (c : unitary ℂ) :
    aeval (residualSubst c) P = P
```

def hyperchargeEigen [source]

Change to hypercharge eigen-coordinates: a_k in terms of z, z^-, w_0, w^-_0, w_1, w^-_1 (indices `0..5`). This diagonalises the gauge-torus rotation into a scaling.

```
noncomputable def hyperchargeEigen : Fin 6 → MvPolynomial (Fin 6) ℂ
```

def diagCartan [source]

The Cartan hypercharge, diagonal in eigen-coordinates: charges `(1,-1,1,-1,-1,1)`.

```
noncomputable def diagCartan (u : unitary ℂ) : Fin 6 → MvPolynomial (Fin 6) ℂ
```

def diagRes [source]

The residual `U(1)`, diagonal in eigen-coordinates: only the perpendicular pair is charged.

```
noncomputable def diagRes (c : unitary ℂ) : Fin 6 → MvPolynomial (Fin 6) ℂ
```

lemma bind₁_diagCartan_hyperchargeEigen [source]

Conjugation identity: the diagonal Cartan scaling, pulled back through the eigen-coordinate change, is the (complexified) Cartan rotation substitution.

```
lemma bind₁_diagCartan_hyperchargeEigen (u : unitary ℂ) (k : Fin 6) :
    bind₁ (diagCartan u) (hyperchargeEigen k)
      = bind₁ hyperchargeEigen (map (algebraMap ℝ ℂ) (cartanSubst u k))
```

lemma bind₁_diagRes_hyperchargeEigen [source]

Conjugation identity for the residual `U(1)`.

```
lemma bind₁_diagRes_hyperchargeEigen (c : unitary ℂ) (k : Fin 6) :
    bind₁ (diagRes c) (hyperchargeEigen k)
      = bind₁ hyperchargeEigen (map (algebraMap ℝ ℂ) (residualSubst c k))
```

def cartanCharge [source]

The Cartan hypercharges of z, z^{-}, w_0, w^{-}_0, w_1, w^{-}_1.

```
def cartanCharge : Fin 6 → ℤ
```

def hyperCharge [source]

The residual-$U(1)$ hypercharges (only the perpendicular pair is charged).

```
def hyperCharge : Fin 6 → ℤ
```

def Qslice [source]

The slice potential, complexified and written in hypercharge eigen-coordinates.

```
noncomputable def Qslice (P : MvPolynomial (Fin 6) ℝ) : MvPolynomial (Fin 6) ℂ
```

lemma diagCartan_eq [source]

The Cartan diagonal in the charge form consumed by the charge-balancing engine.

```
lemma diagCartan_eq (u : unitary ℂ) :
    diagCartan u = fun i => C ((u : ℂ) ^ (cartanCharge i)) * X i
```

lemma diagRes_eq [source]

The residual diagonal in the charge form consumed by the engine.

```
lemma diagRes_eq (c : unitary ℂ) :
    diagRes c = fun i => C (((c : ℂ) ^ 6) ^ (hyperCharge i)) * X i
```

lemma bind₁_diagCartan_Qslice [source]

In eigen-coordinates, the Cartan hypercharge acts by the diagonal scaling, and the slice potential is invariant under it.

```
lemma bind₁_diagCartan_Qslice {V : EffectivePotential} (hI : IsInvariant V)
    {P : MvPolynomial (Fin 6) ℝ} (hP : ∀ a, V (sliceR a) = P.eval a) (u : unitary ℂ) :
    bind₁ (diagCartan u) (Qslice P) = Qslice P
```

lemma bind₁_diagRes_Qslice [source]

Likewise for the residual $U(1)$.

```
lemma bind₁_diagRes_Qslice {V : EffectivePotential} (hI : IsInvariant V)
    {P : MvPolynomial (Fin 6) ℝ} (hP : ∀ a, V (sliceR a) = P.eval a) (c : unitary ℂ) :
    bind₁ (diagRes c) (Qslice P) = Qslice P
```

lemma exists_infiniteOrder_unitary [source]

There is a gauge phase of infinite order (`exp i`), needed to run charge balancing.

```
lemma exists_infiniteOrder_unitary :
    ∃ ω : unitary ℂ, ∀ n : ℤ, (ω : ℂ) ^ n = 1 → n = 0
```

lemma coeff_Qslice_eq_zero [source]

***Hypercharge balancing.** Every monomial of the slice potential `Qslice P` (in eigen-coordinates) that carries nonzero Cartan or residual hypercharge has vanishing coefficient.*

```
lemma coeff_Qslice_eq_zero {V : EffectivePotential} (hI : IsInvariant V)
    {P : MvPolynomial (Fin 6) ℝ} (hP : ∀ a, V (sliceR a) = P.eval a) (m : Fin 6 →₀ ℕ)
    (hm : (∑ i ∈ m.support, (m i : ℤ) * cartanCharge i ≠ 0) ∨
          (∑ i ∈ m.support, (m i : ℤ) * hyperCharge i ≠ 0)) :
    coeff m (Qslice P) = 0
```

def neutralBilinear [source]

The five hypercharge-neutral quadratic bilinears in eigen-coordinates: $z\,z^{-}$, $w_0\,w^{-}_0$, $z\,w^{-}_0$, $z^{-}\,w_0$, $w_1\,w^{-}_1$.

```
noncomputable def neutralBilinear : Fin 5 → MvPolynomial (Fin 6) ℂ
```

lemma charge_univ_eq_support [source]

The charge of a monomial, summed over the whole index set, equals the sum over its support.

```
lemma charge_univ_eq_support (w : Fin 6 → ℤ) (m : Fin 6 →₀ ℕ) :
    ∑ i, (m i : ℤ) * w i = ∑ i ∈ m.support, (m i : ℤ) * w i
```

lemma chargeSum_add [source]

A charge sum is additive in the monomial.

```
lemma chargeSum_add (w : Fin 6 → ℤ) (a b : Fin 6 →₀ ℕ) :
    ∑ k, ((a + b) k : ℤ) * w k = (∑ k, (a k : ℤ) * w k) + ∑ k, (b k : ℤ) * w k
```

lemma chargeSum_single [source]

The charge sum of a single generator is the charge of that variable.

```
lemma chargeSum_single (w : Fin 6 → ℤ) (i : Fin 6) :
    ∑ k, ((Finsupp.single i (1 : ℕ)) k : ℤ) * w k = w i
```

lemma monomial_mem_adjoin_neutralBilinear [source]

***Generation.** Every hypercharge-neutral monomial is a product of the five bilinears.*

```
lemma monomial_mem_adjoin_neutralBilinear (m : Fin 6 →₀ ℕ)
    (hA : ∑ i, (m i : ℤ) * cartanCharge i = 0) (hB : ∑ i, (m i : ℤ) * hyperCharge i = 0) :
```

```
    monomial m (1 : ℂ) ∈ Algebra.adjoin ℂ (Set.range neutralBilinear)
```

lemma Qslice_mem_adjoin_neutralBilinear [source]

The slice potential lies in the subalgebra generated by the five bilinears: every monomial that survives is hypercharge-neutral, hence a product of the bilinears.

```
lemma Qslice_mem_adjoin_neutralBilinear {V : EffectivePotential} (hI : IsInvariant V)
    {P : MvPolynomial (Fin 6) ℝ} (hP : ∀ a, V (sliceR a) = P.eval a) :
    Qslice P ∈ Algebra.adjoin ℂ (Set.range neutralBilinear)
```

lemma exists_aeval_neutralBilinear [source]

Consequently the complexified slice potential is `aeval neutralBilinear G` for some polynomial `G` in the five bilinears.

```
lemma exists_aeval_neutralBilinear {V : EffectivePotential} (hI : IsInvariant V)
    {P : MvPolynomial (Fin 6) ℝ} (hP : ∀ a, V (sliceR a) = P.eval a) :
    ∃ G : MvPolynomial (Fin 5) ℂ, aeval neutralBilinear G = Qslice P
```

def aRep [source]

The slice parameters realising `repHiggs X` as a point of the slice family.

```
def aRep (X : Fin 4 → ℝ) : Fin 6 → ℝ
```

lemma repHiggs_eq_sliceR [source]

```
lemma repHiggs_eq_sliceR (X : Fin 4 → ℝ) : repHiggs X = sliceR (aRep X)
```

def eigenPoint [source]

The hypercharge eigen-point `(z, z̄, w₀, w̄₀, w₁, w̄₁)` of `repHiggs X`: here `z = X₀` is real, `w₀ = X₁ + i X₂` and `w₁ = X₃` is real.

```
noncomputable def eigenPoint (X : Fin 4 → ℝ) : Fin 6 → ℂ
```

lemma aeval_hyperchargeEigen_eigenPoint [source]

The eigen-coordinate change sends the eigen-point of `repHiggs X` back to its slice parameters.

```
lemma aeval_hyperchargeEigen_eigenPoint (X : Fin 4 → ℝ) (k : Fin 6) :
    aeval (eigenPoint X) (hyperchargeEigen k) = algebraMap ℝ ℂ (aRep X k)
```

lemma eval_Qslice_eigenPoint [source]

The complexified slice potential, evaluated at the eigen-point of `repHiggs X`, returns the real value `V (repHiggs X)`.

```
lemma eval_Qslice_eigenPoint (P : MvPolynomial (Fin 6) ℝ) (X : Fin 4 → ℝ) :
    eval (eigenPoint X) (Qslice P) = algebraMap ℝ ℂ (P.eval (aRep X))
```

def realPart [source]

The real part of a complex polynomial, taken coefficient-wise.

```
noncomputable def realPart (H : MvPolynomial (Fin 5) ℂ) : MvPolynomial (Fin 5) ℝ
```

lemma realPart_C [source]

```
lemma realPart_C (a : ℂ) : realPart (C a) = C a.re
```

lemma realPart_add [source]

```
lemma realPart_add (p q : MvPolynomial (Fin 5) ℂ) :
    realPart (p + q) = realPart p + realPart q
```

lemma realPart_mul_X [source]

```
lemma realPart_mul_X (p : MvPolynomial (Fin 5) ℂ) (i : Fin 5) :
    realPart (p * X i) = realPart p * X i
```

lemma realPart_eval [source]

Evaluating a complex polynomial at a real point and taking the real part is the same as evaluating its real part.

```
lemma realPart_eval (H : MvPolynomial (Fin 5) ℂ) (y : Fin 5 → ℝ) :
    (eval (fun j => (↑(y j) : ℂ)) H).re = (realPart H).eval y
```

def sliceBilinear [source]

The five real gauge-invariant bilinears, evaluated at `repHiggs X`: $\|\Phi 1\|^2$, $Re\langle\langle\rangle\rangle$, $Im\langle\langle\rangle\rangle$, $|\Phi 2_0|^2$, $|\Phi 2_1|^2$.

```
def sliceBilinear (X : Fin 4 → ℝ) : Fin 5 → ℝ
```

def transf [source]

The complex substitution expressing each bilinear, at the eigen-point, through the real generators (the off-diagonal pair $z\,\bar{w}_0$, $\bar{z}\,w_0$ mix $Re\langle\langle\rangle\rangle$ and $Im\langle\langle\rangle\rangle$).

```
noncomputable def transf : Fin 5 → MvPolynomial (Fin 5) ℂ
```

lemma neutralBilinear_eval_eigenPoint [source]

The bilinears at the eigen-point of `repHiggs X` are the real generators, read through `transf`.

```
lemma neutralBilinear_eval_eigenPoint (X : Fin 4 → ℝ) (i : Fin 5) :
    eval (eigenPoint X) (neutralBilinear i)
      = eval (fun j => (↑(sliceBilinear X j) : ℂ)) (transf i)
```

lemma exists_polynomial_repHiggs_sliceBilinear [source]

***The potential on representatives, in the bilinears.** A gauge-invariant polynomial potential, on the representative family, is a polynomial in the five real gauge-invariant bilinears $\|\Phi 1\|^2$, $Re\langle\langle\rangle\rangle$, $Im\langle\langle\rangle\rangle$, $|\Phi 2_0|^2$, $|\Phi 2_1|^2$.*

```
lemma exists_polynomial_repHiggs_sliceBilinear {V : EffectivePotential} {n : ℕ}
    (hI : IsInvariant V) (h : HasMaxMassDimLE V n) :
    ∃ p : MvPolynomial (Fin 5) ℝ, ∀ X : Fin 4 → ℝ, V (repHiggs X) = p.eval (sliceBilinear X)
```

def sliceBilinearPoly [source]

The five bilinear generators, as polynomials in the four representative parameters.

```
noncomputable def sliceBilinearPoly : Fin 5 → MvPolynomial (Fin 4) ℝ
```

def gramPoly [source]

The four Gram components, as polynomials in the four representative parameters.

```
noncomputable def gramPoly : Fin 1 ⊕ Fin 3 → MvPolynomial (Fin 4) ℝ
```

lemma exists_clearing_mem [source]

Some power of $\|\Phi 1\|^2 = X_0^2$ times the value polynomial lies in the Gram subalgebra: multiplying by X_0^2 pairs each $X_1^2+X_2^2$ into $(X_0X_1)^2+(X_0X_2)^2$ and each X_3^2 into the determinant $X_0^2X_3^2$, both of which are Gram polynomials.

```
lemma exists_clearing_mem (p : MvPolynomial (Fin 5) ℝ) :
    ∃ N : ℕ, (X 0) ^ (2 * N) * aeval sliceBilinearPoly p ∈
      Algebra.adjoin ℝ (Set.range gramPoly)
```

lemma exists_gram_clearing [source]

***Denominator clearing.** For the value polynomial p, some power of $\|\Phi 1\|^2 = X_0^2$ times $p \circ sliceBilinear$ is a polynomial in the Gram vector.*

```
lemma exists_gram_clearing (p : MvPolynomial (Fin 5) ℝ) :
    ∃ (A : MvPolynomial (Fin 1 ⊕ Fin 3) ℝ) (N : ℕ), ∀ X : Fin 4 → ℝ,
      (X 0) ^ (2 * N) * p.eval (sliceBilinear X) = A.eval ((repHiggs X).gramVector)
```

lemma exists_normSq_Φ1_clearing [source]

***$\|\Phi 1\|^2$-clearing, on all configurations.** A power of $\|\Phi 1\|^2$ times a gauge-invariant polynomial potential is, everywhere, a polynomial in the Gram vector.*

```
lemma exists_normSq_Φ1_clearing {V : EffectivePotential} {n : ℕ}
    (hI : IsInvariant V) (h : HasMaxMassDimLE V n) :
    ∃ (A : MvPolynomial (Fin 1 ⊕ Fin 3) ℝ) (N : ℕ), ∀ φ : TwoHiggsDoublet,
      (‖φ.Φ1‖ ^ 2) ^ N * V φ = A.eval φ.gramVector
```

def swapSubst [source]

The Gram-vector substitution induced by swapping the doublets (sign flip on the imaginary and difference components).

```
noncomputable def swapSubst : (Fin 1 ⊕ Fin 3) → MvPolynomial (Fin 1 ⊕ Fin 3) ℝ
```

lemma exists_normSq_Φ2_clearing [source]

***‖Φ2‖²-clearing, on all configurations.** A power of ‖Φ2‖² times a gauge-invariant polynomial potential is, everywhere, a polynomial in the Gram vector. Obtained from `exists_normSq_Φ1_clearing` for the doublet-swapped potential.*

```
lemma exists_normSq_Φ2_clearing {V : EffectivePotential} {n : ℕ}
    (hI : IsInvariant V) (h : HasMaxMassDimLE V n) :
    ∃ (B : MvPolynomial (Fin 1 ⊕ Fin 3) ℝ) (M : ℕ), ∀ φ : TwoHiggsDoublet,
      (‖φ.Φ2‖ ^ 2) ^ M * V φ = B.eval φ.gramVector
```

lemma gramPoly_injective [source]

The four Gram invariants are algebraically independent: the Gram substitution is injective.

```
lemma gramPoly_injective :
    Function.Injective
      (aeval gramPoly : MvPolynomial (Fin 1 ⊕ Fin 3) ℝ → MvPolynomial (Fin 4) ℝ)
```

lemma uPow_dvd [source]

‖Φ1‖² and ‖Φ2‖², as the distinct linear forms `(g₀±g₃)/2` of the Gram ring, are coprime: if ‖Φ1‖^2N · B = ‖Φ2‖^2M · A then ‖Φ1‖^2N ∣ A.

```
lemma uPow_dvd {N M : ℕ} {A B : MvPolynomial (Fin 1 ⊕ Fin 3) ℝ}
    (hAB : (C (1 / 2) * (X (Sum.inl 0) + X (Sum.inr 2))) ^ N * B
       = (C (1 / 2) * (X (Sum.inl 0) - X (Sum.inr 2))) ^ M * A) :
    (C (1 / 2) * (X (Sum.inl 0) + X (Sum.inr 2))) ^ N ∣ A
```

lemma exists_polynomial_on_repHiggs [source]

***On representatives.** A gauge-invariant polynomial potential, restricted to the polynomial family of orbit representatives `repHiggs X`, is a polynomial in the Gram components of that family. Proved by the doublet-swap argument: clearing the ‖Φ1‖² factor (aligning Φ1) and the ‖Φ2‖² factor (aligning Φ2, via the gauge-commuting swap), then using that ‖Φ1‖² and ‖Φ2‖² are coprime in the Gram ring.*

```
lemma exists_polynomial_on_repHiggs {V : EffectivePotential} {n : ℕ}
    (hI : IsInvariant V) (h : HasMaxMassDimLE V n) :
    ∃ p : MvPolynomial (Fin 1 ⊕ Fin 3) ℝ,
      ∀ X : Fin 4 → ℝ, V (repHiggs X) = p.eval (repHiggs X).gramVector
```

lemma effectivePotential_is_polynomial_gramVector [source]

***The two Higgs doublet potential in the bilinear formalism.** A gauge-invariant polynomial effective potential of maximum mass dimension `n` is a polynomial in the four gauge-invariant bilinears — the entries of the Gram vector.*

```
lemma effectivePotential_is_polynomial_gramVector {V : EffectivePotential} {n : ℕ}
    (hI: IsInvariant V) (h : HasMaxMassDimLE V n) :
    ∃ p : MvPolynomial (Fin 1 ⊕ Fin 3) ℝ, (∀ φ : TwoHiggsDoublet, V φ = p.eval φ.gramVector)
```

2.106 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.Module

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/Module.lean on GitHub.

instance Add TwoHiggsDoublet [source]

```
instance : Add TwoHiggsDoublet where
```

lemma add_fst [source]

```
@[simp]
lemma add_fst (H1 H2 : TwoHiggsDoublet) : (H1 + H2).Φ1 = H1.Φ1 + H2.Φ1
```

lemma add_snd [source]

```
@[simp]
lemma add_snd (H1 H2 : TwoHiggsDoublet) : (H1 + H2).Φ2 = H1.Φ2 + H2.Φ2
```

instance Zero TwoHiggsDoublet [source]

```
instance : Zero TwoHiggsDoublet where
```

lemma zero_fst [source]

```
@[simp]
lemma zero_fst : (0 : TwoHiggsDoublet).Φ1 = 0
```

lemma zero_snd [source]

```
@[simp]
lemma zero_snd : (0 : TwoHiggsDoublet).Φ2 = 0
```

instance SMul ℂ TwoHiggsDoublet [source]

```
instance : SMul ℂ TwoHiggsDoublet where
```

lemma smul_fst [source]

```
@[simp]
lemma smul_fst (c : ℂ) (H : TwoHiggsDoublet) : (c • H).Φ1 = c • H.Φ1
```

lemma smul_snd [source]

```
@[simp]
lemma smul_snd (c : ℂ) (H : TwoHiggsDoublet) : (c • H).Φ2 = c • H.Φ2
```

instance Neg TwoHiggsDoublet [source]

```
instance : Neg TwoHiggsDoublet where
```

lemma neg_fst [source]

```
@[simp]
lemma neg_fst (H : TwoHiggsDoublet) : (-H).Φ1 = -H.Φ1
```

lemma neg_snd [source]

```
@[simp]
lemma neg_snd (H : TwoHiggsDoublet) : (-H).Φ2 = -H.Φ2
```

instance AddCommGroup TwoHiggsDoublet [source]

```
instance : AddCommGroup TwoHiggsDoublet where
```

instance Module ℂ TwoHiggsDoublet [source]

```
instance : Module ℂ TwoHiggsDoublet where
```

2.107 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.OrbitRepresentative

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/OrbitRepresentative.lean on GitHub.

def repHiggs [source]

A four–real–parameter polynomial family of representatives for the gauge orbits: the first doublet is `(X₀, 0)` and the second is `(X₁ + i X₂, X₃)`.

```
def repHiggs (X : Fin 4 → ℝ) : TwoHiggsDoublet where
```

lemma repHiggs_Φ1 [source]

```
@[simp]
lemma repHiggs_Φ1 (X : Fin 4 → ℝ) : (repHiggs X).Φ1 = ![(X 0 : ℂ), 0]
```

lemma repHiggs_Φ2 [source]

```
@[simp]
lemma repHiggs_Φ2 (X : Fin 4 → ℝ) :
    (repHiggs X).Φ2 = ![(X 1 : ℂ) + Complex.I * (X 2 : ℂ), (X 3 : ℂ)]
```

lemma normSq_repHiggs_Φ1 [source]

```
lemma normSq_repHiggs_Φ1 (X : Fin 4 → ℝ) : ‖(repHiggs X).Φ1‖ ^ 2 = X 0 ^ 2
```

lemma normSq_repHiggs_Φ2 [source]

```
lemma normSq_repHiggs_Φ2 (X : Fin 4 → ℝ) :
    ‖(repHiggs X).Φ2‖ ^ 2 = X 1 ^ 2 + X 2 ^ 2 + X 3 ^ 2
```

lemma inner_repHiggs [source]

```
lemma inner_repHiggs (X : Fin 4 → ℝ) :
    ⟪(repHiggs X).Φ1, (repHiggs X).Φ2⟫_ℂ = (X 0 : ℂ) * ((X 1 : ℂ) + Complex.I * (X 2 : ℂ))
```

lemma gramVector_repHiggs_inl [source]

```
@[simp]
lemma gramVector_repHiggs_inl (X : Fin 4 → ℝ) :
    (repHiggs X).gramVector (Sum.inl 0) = X 0 ^ 2 + (X 1 ^ 2 + X 2 ^ 2 + X 3 ^ 2)
```

lemma gramVector_repHiggs_inr0 [source]

```
@[simp]
lemma gramVector_repHiggs_inr0 (X : Fin 4 → ℝ) :
    (repHiggs X).gramVector (Sum.inr 0) = 2 * X 0 * X 1
```

lemma gramVector_repHiggs_inr1 [source]

```
@[simp]
lemma gramVector_repHiggs_inr1 (X : Fin 4 → ℝ) :
    (repHiggs X).gramVector (Sum.inr 1) = 2 * X 0 * X 2
```

lemma gramVector_repHiggs_inr2 [source]

```
@[simp]
lemma gramVector_repHiggs_inr2 (X : Fin 4 → ℝ) :
    (repHiggs X).gramVector (Sum.inr 2) = X 0 ^ 2 - (X 1 ^ 2 + X 2 ^ 2 + X 3 ^ 2)
```

lemma exists_smul_eq_repHiggs [source]

Every field configuration is in the gauge orbit of a representative `repHiggs X`. This is the key fact that lets a gauge invariant potential be evaluated on the polynomial family.

```
lemma exists_smul_eq_repHiggs (φ : TwoHiggsDoublet) :
    ∃ (X : Fin 4 → ℝ) (g : GaugeGroupI), g • φ = repHiggs X
```

2.108 PhyslibAlpha.Particles.BeyondTheStandardModel.TwoHDM.SwapDoublet

PhyslibAlpha/Particles/BeyondTheStandardModel/TwoHDM/SwapDoublet.lean on GitHub.

def swapDoublet [source]

Swapping the two doublets, as an $\mathbb{R}$-linear map. It commutes with the gauge action, so it sends gauge-invariant polynomial potentials to gauge-invariant polynomial potentials, but turns the alignment of `Φ1` into the alignment of `Φ2`.

```
def swapDoublet : TwoHiggsDoublet →ₗ[ℝ] TwoHiggsDoublet where
```

lemma swapDoublet_smul [source]

```
lemma swapDoublet_smul (g : StandardModel.GaugeGroupI) (φ : TwoHiggsDoublet) :
    swapDoublet (g • φ) = g • swapDoublet φ
```

lemma gramVector_swapDoublet_inl [source]

Swapping the doublets sends the gram vector through the sign flip of the imaginary and difference components.

```
lemma gramVector_swapDoublet_inl (φ : TwoHiggsDoublet) :
    (swapDoublet φ).gramVector (Sum.inl 0) = φ.gramVector (Sum.inl 0)
```

lemma gramVector_swapDoublet_inr0 [source]

```
lemma gramVector_swapDoublet_inr0 (φ : TwoHiggsDoublet) :
    (swapDoublet φ).gramVector (Sum.inr 0) = φ.gramVector (Sum.inr 0)
```

lemma gramVector_swapDoublet_inr1 [source]

```
lemma gramVector_swapDoublet_inr1 (φ : TwoHiggsDoublet) :
    (swapDoublet φ).gramVector (Sum.inr 1) = -φ.gramVector (Sum.inr 1)
```

lemma gramVector_swapDoublet_inr2 [source]

```
lemma gramVector_swapDoublet_inr2 (φ : TwoHiggsDoublet) :
    (swapDoublet φ).gramVector (Sum.inr 2) = -φ.gramVector (Sum.inr 2)
```

lemma HasMaxMassDimLE.comp_swapDoublet [source]

```
lemma HasMaxMassDimLE.comp_swapDoublet {V : EffectivePotential} {n : ℕ}
    (h : HasMaxMassDimLE V n) : HasMaxMassDimLE (fun φ => V (swapDoublet φ)) n
```

lemma IsInvariant.comp_swapDoublet [source]

```
lemma IsInvariant.comp_swapDoublet {V : EffectivePotential} (hI : IsInvariant V) :
    IsInvariant (fun φ => V (swapDoublet φ))
```

2.109 QuantumInfo.ClassicalInfo.Prob

QuantumInfo/ClassicalInfo/Prob.lean on GitHub.

theorem mix_one [source]

```
@[simp]
theorem mix_one [inst : Mixable U T] (x₁ x₂ : T) : (1 : Prob) [ x₁ ↔ x₂ : inst] = x₁
```

2.110 QuantumInfo.Entropy.DPI

QuantumInfo/Entropy/DPI.lean on GitHub.

theorem sandwichedRelRentropy_tendsto_qRelativeEnt [source]

As `α → 1⁺`, the sandwiched Rényi relative entropy `D̃_α(ρ‖σ)` tends to the relative entropy `𝐃(ρ‖σ) = D̃_1(ρ‖σ)`, by continuity of `α ↦ D̃_α` on `(0, ∞)`.

```
theorem sandwichedRelRentropy_tendsto_qRelativeEnt (ρ σ : MState d) :
    Filter.Tendsto (fun α : ℝ => D̃ α(ρ‖σ)) (𝓝[>] 1) (𝓝 𝐃(ρ‖σ))
```

lemma sandwichedTraceFunctional_sub_one_div_eventually_le [source]

As `α → 1⁺`, the difference quotient `(Q̃_α(ρ‖σ) - 1) / (α - 1)` is eventually bounded above by a function tending to `𝐃(ρ‖σ).toReal`. This is the key estimate for transferring joint convexity of the trace functional `Q̃_α` to the relative entropy `𝐃` in `qRelativeEnt_joint_convexity`.

```
private lemma sandwichedTraceFunctional_sub_one_div_eventually_le
    (ρ σ : MState d) (hker : σ.M.ker ≤ ρ.M.ker) :
    ∃ u : ℝ → ℝ, Filter.Tendsto u (𝓝[>] 1) (𝓝 (𝐃(ρ‖σ)).toReal) ∧
      ∀ᶠ α in 𝓝[>] 1, (Q̃ α(ρ‖σ) - 1) / (α - 1) ≤ u α
```

lemma mix_M_eq_weighted_sum [source]

A binary `Mixable` mixture of states, written as a weighted sum of matrices over `Fin 2` — the form consumed by `sandwichedTraceFunctional_jointly_convex` and `HermitianMat.ker_weighted_sum_le`.

```
private lemma mix_M_eq_weighted_sum (p : Prob) (τ₁ τ₂ : MState d) :
    (p [τ₁ ↔ τ₂]).M = ∑ i, ![(p : ℝ), 1 - (p : ℝ)] i • (![τ₁, τ₂] i).M
```

lemma ker_mix_le [source]

A binary mixture preserves the support condition (kernel inclusion) of its components.

```
private lemma ker_mix_le (p : Prob) {ρ₁ ρ₂ σ₁ σ₂ : MState d}
    (hker₁ : σ₁.M.ker ≤ ρ₁.M.ker) (hker₂ : σ₂.M.ker ≤ ρ₂.M.ker) :
    (p [σ₁ ↔ σ₂]).M.ker ≤ (p [ρ₁ ↔ ρ₂]).M.ker
```

lemma sandwichedTraceFunctional_mix_le [source]

Binary case of the joint convexity of the trace functional `Q̃_α` for `α > 1` (`sandwichedTraceFunctional_jointl` stated for a `Mixable` mixture.

```
private lemma sandwichedTraceFunctional_mix_le (hα : 1 < α) (p : Prob)
    {ρ₁ ρ₂ σ₁ σ₂ : MState d}
    (hker₁ : σ₁.M.ker ≤ ρ₁.M.ker) (hker₂ : σ₂.M.ker ≤ ρ₂.M.ker) :
    Q̃ α(p [ρ₁ ↔ ρ₂]‖p [σ₁ ↔ σ₂]) ≤
      (p : ℝ) * Q̃ α(ρ₁‖σ₁) + (1 - (p : ℝ)) * Q̃ α(ρ₂‖σ₂)
```

2.111 QuantumInfo.Entropy.Relative

QuantumInfo/Entropy/Relative.lean on GitHub.

theorem qRelativeEnt_ne_top_iff [source]

The quantum relative entropy is finite exactly when the support condition `σ.M.ker ≤ ρ.M.ker` holds.

```
theorem qRelativeEnt_ne_top_iff {ρ σ : MState d} : D(ρ‖σ) ≠ ⊤ ↔ σ.M.ker ≤ ρ.M.ker
```

theorem qRelativeEnt_eq_top_iff [source]

The quantum relative entropy is `⊤` exactly when the support condition fails.

```
theorem qRelativeEnt_eq_top_iff {ρ σ : MState d} : D(ρ‖σ) = ⊤ ↔ ¬ σ.M.ker ≤ ρ.M.ker
```

2.112 scripts.autolabel

scripts/autolabel.lean on GitHub.

def MAX_LABELS [source]

Maximal number of labels which can be added. If more are applicable, nothing will be added.

```
def MAX_LABELS
```

inductive Label [source]

Physlib's Github topic labels

```
inductive Label where
  | «t-classical-field-theory»
  | «t-classical-mechanics»
  | «t-condensed-matter»
  | «t-cosmology»
  | «t-electromagnetism»
  | «t-fluid-dynamics»
  | «t-mathematics»
  | «t-meta»
  | «t-optics»
  | «t-particles»
  | «t-qft»
  | «t-quantum-mechanics»
  | «t-relativity»
  | «t-space-and-time»
  | «t-statistical-mechanics»
  | «t-string-theory»
  | «t-thermodynamics»
  | «t-units»
  | «t-capacity-qi»
  | «t-channels-qi»
  | «t-classical-info-qi»
  | «t-entropy-qi»
  | «t-for-mathlib-qi»
  | «t-measurements-qi»
  | «t-operators-qi»
  | «t-resource-theory-qi»
  | «t-states-qi»
  | «CI»
  deriving BEq, Hashable, Repr
```

def physlibLabels [source]

```
def physlibLabels : Array Label
```

def toString [source]

```
def Label.toString : Label → String
  | .«t-classical-field-theory»                  => "t-classical-field-theory"
  | .«t-classical-mechanics»         => "t-classical-mechanics"
  | .«t-condensed-matter»          => "t-condensed-matter"
  | .«t-cosmology»                    => "t-cosmology"
  | .«t-electromagnetism»             => "t-electromagnetism"
  | .«t-fluid-dynamics»             => "t-fluid-dynamics"
  | .«t-mathematics»               => "t-mathematics"
  | .«t-meta»                       => "t-meta"
  | .«t-optics»                     => "t-optics"
  | .«t-particles»                  => "t-particles"
  | .«t-qft»                        => "t-qft"
  | .«t-quantum-mechanics»          => "t-quantum-mechanics"
  | .«t-relativity»                => "t-relativity"
  | .«t-space-and-time»             => "t-space-and-time"
  | .«t-statistical-mechanics»      => "t-statistical-mechanics"
  | .«t-string-theory»              => "t-string-theory"
  | .«t-thermodynamics»             => "t-thermodynamics"
  | .«t-units»                      => "t-units"
  | .«t-capacity-qi»                   => "t-capacity-qi"
  | .«t-channels-qi»                   => "t-channels-qi"
  | .«t-classical-info-qi»             => "t-classical-info-qi"
  | .«t-entropy-qi»                    => "t-entropy-qi"
  | .«t-for-mathlib-qi»                => "t-for-mathlib-qi"
  | .«t-measurements-qi»               => "t-measurements-qi"
  | .«t-operators-qi»                  => "t-operators-qi"
  | .«t-resource-theory-qi»            => "t-resource-theory-qi"
  | .«t-states-qi»                     => "t-states-qi"
  | .«CI»                              => "CI"
```

instance ToString Label [source]

```
instance : ToString Label where
```

structure LabelData [source]

*A `LabelData` consists of the * The `dirs` field is the array of all "root paths" such that a modification in a file contained in one of these paths should be labelled with `label`. * The `exclusions` field is the array of all "root paths" that are excluded, among the ones that start with the ones in `dirs`. Any modifications to a file in an excluded path is ignored for the purposes of labelling. * The `dependencies` field is the array of all labels, which are lower in the import hierarchy and which should be excluded if the label is present.*

```
structure LabelData (label : Label) where
  dirs : Array FilePath := if label.toString.startsWith "t-" && label.toString.endsWith "-qi"
    then
      #["QuantumInfo" / ("".intercalate (label.toString.splitOn "-" |>.drop 1 |>.dropLast |>.
    map .capitalize))]
  else if label.toString.startsWith "t-" then
```

```
    #["Physlib" / ("".intercalate (label.toString.splitOn "-" |>.drop 1 |>.map .capitalize))]
    else #[]
  exclusions : Array FilePath := #[]
  dependencies : Array Label := #[]
  deriving BEq, Hashable
```

def physlibLabelData [source]

This function maps each label to the corresponding paths For now, no exclusions or dependencies are specified -

```
def physlibLabelData: (l: Label) → LabelData l
  | .«t-classical-field-theory» => {}
  | .«t-classical-mechanics» => {}
  | .«t-condensed-matter» => {}
  | .«t-cosmology» => {}
  | .«t-electromagnetism» => {}
  | .«t-fluid-dynamics» => {}
  | .«t-mathematics» => {}
  | .«t-meta» => {}
  | .«t-optics» => {}
  | .«t-particles» => {}
  | .«t-qft» => { dirs := #["Physlib" / "QFT"] }
  | .«t-quantum-mechanics» => {}
  | .«t-relativity» => {}
  | .«t-space-and-time» => {}
  | .«t-statistical-mechanics» => {}
  | .«t-string-theory» => {}
  | .«t-thermodynamics» => {}
  | .«t-units» => {}
  | .«t-capacity-qi» => {}
  | .«t-channels-qi» => {}
  | .«t-classical-info-qi» => {}
  | .«t-entropy-qi» => {}
  | .«t-for-mathlib-qi» => {}
  | .«t-measurements-qi» => {}
  | .«t-operators-qi» => {}
  | .«t-resource-theory-qi» => {}
  | .«t-states-qi» => {}
  | .«CI» => { dirs := #["scripts"] }
```

def physlibUnlabelled [source]

Exceptions inside `Physlib/` which are not covered by any label. (For the First versions, no exceptions)

```
def physlibUnlabelled : Array FilePath
```

def quantuminfoUnlabelled [source]

Exceptions inside `QuantumInfo/` which are not covered by any label. (For the First versions, no exceptions)

```
def quantuminfoUnlabelled : Array FilePath
```

def isPrefixOf [source]

Checks if the folder `path` lies inside the folder `dir`. Copied verbatim from mathlib `scripts/autolabel.lean`

```
def _root_.System.FilePath.isPrefixOf (dir path : FilePath) : Bool
```

def getMatchingLabels [source]

*Return all labels in `physlibLabels` which match at least one of the `files`. * `files`: array of relative paths starting from the physlib root directory.*

```
def getMatchingLabels (files : Array FilePath) : Array Label
```

def collectLabelsAndDependentLabels [source]

Helper function: union of all labels and all their dependent labels

```
partial def collectLabelsAndDependentLabels (labels: Array Label) : Array Label
```

def dropDependentLabels [source]

Reduce a list of labels to not include any which are dependencies of other labels in the list

```
def dropDependentLabels (labels: Array Label) : Array Label
```

def findUncoveredPaths [source]

Testing function to ensure the labels defined in `physlibLabels` cover all subfolders of `Physlib/` and `QuantumInfo/`.

```
partial def findUncoveredPaths (path : FilePath) (exceptions : Array FilePath := #[]) :
    IO <| Array FilePath
```

def githubAnnotation [source]

Create a message which GitHub CI parses as annotation and displays at the specified file.

Note: `file` is duplicated below so that it is also visible in the plain text output.

** `type`: "error" or "warning" * `file`: file where the annotation should be displayed * `title`: title of the annotation * `message`: annotation message*

```
def githubAnnotation (type file title message : String) : String
```

def main [source]

`args` is expected to have length 0 or 1, where the first argument is the PR number.

If a PR number is provided, the script requires GitHub CLI `gh` to be installed in order to add the label to the PR.

Exit codes:

- `0`: success - `1`: invalid arguments provided - `2`: invalid labels defined - `3`: ~labels do not cover all of `Physlib/`~ (unused; only emitting warning)

```
unsafe def main (args : List String): IO UInt32
```