Physlib Monthly Report

1–31 October 2025

Contributors: CasualPhysicsEnjoier, Jard Nijholt, Joseph Tooby-Smith, luisgbariuan, Matteo Cipollina, Prabhoda Chandra Sarjapur, Ronan Lamy

Reviewers: damiano, Joseph Tooby-Smith

Generated 14 August 2026 from commit `c6ebae9`.

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 October 2025 on branch `master` of `leanprover-community/physlib`, between commits `9a8fe27` and `c6ebae9`. Over that period 7 contributors landed 56 commits, changing 25,119 lines across 195 files and adding 510 new Lean declarations, each catalogued with its statement in Section 2. The complete diff is available at GitHub compare.

Contents

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

2 Details by file **3**
- 2.1 `PhysLean.ClassicalMechanics.HarmonicOscillator.Basic` . . . 3
- 2.2 `PhysLean.ClassicalMechanics.RigidBody.Basic` . . . 4
- 2.3 `PhysLean.ClassicalMechanics.VectorFields` . . . 9
- 2.4 `PhysLean.ClassicalMechanics.WaveEquation.Basic` . . . 9
- 2.5 `PhysLean.Cosmology.FLRW.Basic` . . . 11
- 2.6 `PhysLean.Electromagnetism.Distributions.Potential` . . . 13
- 2.7 `PhysLean.Electromagnetism.Dynamics.Basic` . . . 15
- 2.8 `PhysLean.Electromagnetism.Dynamics.CurrentDensity` . . . 16
- 2.9 `PhysLean.Electromagnetism.Dynamics.Hamiltonian` . . . 18
- 2.10 `PhysLean.Electromagnetism.Dynamics.IsExtrema` . . . 20
- 2.11 `PhysLean.Electromagnetism.Dynamics.KineticTerm` . . . 21
- 2.12 `PhysLean.Electromagnetism.Dynamics.Lagrangian` . . . 23
- 2.13 `PhysLean.Electromagnetism.Kinematics.Boosts` . . . 26
- 2.14 `PhysLean.Electromagnetism.Kinematics.ElectricField` . . . 26
- 2.15 `PhysLean.Electromagnetism.Kinematics.EMPotential` . . . 28
- 2.16 `PhysLean.Electromagnetism.Kinematics.FieldStrength` . . . 29
- 2.17 `PhysLean.Electromagnetism.Kinematics.MagneticField` . . . 30
- 2.18 `PhysLean.Electromagnetism.Kinematics.ScalarPotential` . . . 32
- 2.19 `PhysLean.Electromagnetism.Kinematics.VectorPotential` . . . 33

2.20 PhysLean.Electromagnetism.PointParticle.OneDimension . . . 34
2.21 PhysLean.Electromagnetism.Vacuum.Constant . . . 35
2.22 PhysLean.Mathematics.RatComplexNum . . . 36
2.23 PhysLean.Mathematics.VariationalCalculus.HasVarAdjDeriv . . . 37
2.24 PhysLean.Mathematics.VariationalCalculus.HasVarGradient . . . 37
2.25 PhysLean.Meta.AllFilePaths . . . 37
2.26 PhysLean.Meta.Basic . . . 38
2.27 PhysLean.Meta.Sorry . . . 38
2.28 PhysLean.Particles.NeutrinoPhysics.Basic . . . 39
2.29 PhysLean.Particles.SuperSymmetry.SU5.ChargeSpectrum.Basic . . . 41
2.30 PhysLean.Relativity.LorentzGroup.Boosts.Basic . . . 41
2.31 PhysLean.Relativity.SpeedOfLight . . . 43
2.32 PhysLean.Relativity.Tensors.Product . . . 43
2.33 PhysLean.Relativity.Tensors.RealTensor.Vector.Basic . . . 44
2.34 PhysLean.SpaceAndTime.Space.Basic . . . 45
2.35 PhysLean.SpaceAndTime.Space.Distributions.Basic . . . 45
2.36 PhysLean.SpaceAndTime.Space.Distributions.ConstantTime . . . 47
2.37 PhysLean.SpaceAndTime.Space.VectorIdentities . . . 50
2.38 PhysLean.SpaceAndTime.SpaceTime.Basic . . . 51
2.39 PhysLean.SpaceAndTime.SpaceTime.Boosts . . . 52
2.40 PhysLean.SpaceAndTime.SpaceTime.Distributions . . . 52
2.41 PhysLean.SpaceAndTime.SpaceTime.TimeSlice . . . 53
2.42 PhysLean.SpaceAndTime.Time.Basic . . . 54
2.43 PhysLean.StatisticalMechanics.CanonicalEnsemble.Basic . . . 54
2.44 PhysLean.StatisticalMechanics.CanonicalEnsemble.Finite . . . 54
2.45 PhysLean.StatisticalMechanics.CanonicalEnsemble.Lemmas . . . 56
2.46 PhysLean.StringTheory.FTheory.SU5.Charges.Viable . . . 57
2.47 PhysLean.StringTheory.FTheory.SU5.Fluxes.Basic . . . 58
2.48 PhysLean.StringTheory.FTheory.SU5.Fluxes.NoExotics.Completeness . . . 59
2.49 PhysLean.StringTheory.FTheory.SU5.Quanta.Basic . . . 61
2.50 PhysLean.StringTheory.FTheory.SU5.Quanta.FiveQuanta . . . 62
2.51 PhysLean.StringTheory.FTheory.SU5.Quanta.IsViable . . . 67
2.52 PhysLean.StringTheory.FTheory.SU5.Quanta.TenQuanta . . . 68
2.53 PhysLean.Thermodynamics.Temperature.Basic . . . 72
2.54 PhysLean.Units.Basic . . . 74
2.55 PhysLean.Units.Integral . . . 74
2.56 PhysLean.Units.UnitDependent . . . 74
2.57 PhysLean.Units.WithDim.Basic . . . 74
2.58 scripts.MetaPrograms.redundant_imports . . . 75
2.59 scripts.MetaPrograms.runPhysLeanLinters . . . 75
2.60 scripts.MetaPrograms.spelling . . . 75

1 Introduction

During Oct 2025, 7 contributors submitted 56 commits that changed 25,119 lines (+18,014 / -7,105) across 195 files spanning 24 top-level areas of the repository. Of those, 24 files were newly added and 6 were removed outright. This report catalogues 510 newly added Lean declarations: 351 lemmas, 92 defs, 23 informal lemmas, 23 instances, 7 informal definitions, 4 theorems, 4 abbrevs, 4 structures, and 2 classs. We're delighted to recognize the following first-time contributors this month: Jard Nijholt, luisgbariuan, Prabhoda Chandra Sarjapur, and Ronan Lamy.

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
CasualPhysicsEnjoier	329	7
Jard Nijholt	87	1
Joseph Tooby-Smith	26,020	25
luisgbariuan	122	2
Matteo Cipollina	1,627	18
Prabhoda Chandra Sarjapur	135	2
Ronan Lamy	163	1

1.2 Reviewers

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

Reviewer	PRs Reviewed
damiano	2
Joseph Tooby-Smith	6

1.3 Modified subfolders

Subfolder	Files	New	Deleted	Additions	Deletions
`PhysLean/Electromagnetism`	31	+15	-6	+5,854	-2,879
`PhysLean/StringTheory`	11	–	–	+3,134	-2,327
`scripts/MetaPrograms`	11	+4	–	+3,206	-26
`PhysLean/SpaceAndTime`	13	+2	–	+2,457	-359
`PhysLean/Relativity`	34	+1	–	+1,002	-661
`PhysLean/ClassicalMechanics`	6	–	–	+652	-237
`PhysLean/StatisticalMechanics`	5	–	–	+421	-64
`PhysLean/Particles`	23	+1	–	+233	-88
`PhysLean/Meta`	6	+1	–	+248	-58
`PhysLean/Mathematics`	17	–	–	+171	-90
`PhysLean/CondensedMatter`	1	–	–	+197	-8
`PhysLean/Units`	11	–	–	+77	-114

scripts	2	–	–	+54	-103
PhysLean/Cosmology	1	–	–	+98	-24
PhysLean/Thermodynamics	1	–	–	+112	-2
PhysLean/QFT	9	–	–	+25	-24
PhysLean.lean	1	–	–	+27	-13
lake-manifest.json	1	–	–	+15	-15
lakefile.toml	1	–	–	+16	-3
PhysLean/QuantumMechanics	5	–	–	+5	-5
.github/workflows	2	–	–	+7	-2
PhysLean/Optics	1	–	–	+1	-1
README.md	1	–	–	+1	-1
lean-toolchain	1	–	–	+1	-1

1.4 New declarations by kind

Kind	Count
lemma	351
def	92
informal_lemma	23
instance	23
informal_definition	7
theorem	4
abbrev	4
structure	4
class	2

2 Details by file

2.1 PhysLean.ClassicalMechanics.HarmonicOscillator.Basic

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

def gradLagrangian [source]

The Euler-Lagrange operator for the classical harmonic oscillator.

```
noncomputable def gradLagrangian (xₜ : Time → Space 1) : Time → Space 1
```

lemma gradLagrangian_eq_eulerLagrangeOp [source]

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

lemma equationOfMotion_iff_gradLagrangian_zero [source]

```
lemma equationOfMotion_iff_gradLagrangian_zero (xₜ : Time → Space 1) :
    S.EquationOfMotion xₜ ↔ S.gradLagrangian xₜ = 0
```

lemma gradLagrangian_eq_force [source]

The Euler lagrange operator corresponds to Newton's second law.

```
lemma gradLagrangian_eq_force (xₜ : Time → Space 1) (hx : ContDiff ℝ ∞ xₜ) :
    S.gradLagrangian xₜ = fun t => force S (xₜ t) - S.m • ∂ₜ (∂ₜ xₜ) t
```

2.2 PhysLean.ClassicalMechanics.RigidBody.Basic

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

informal_definition coordinate_system [source]

One can describe the motion of rigid body with a fixed (inertial) coordinate system (X,Y,Z) and a moving system (x_1,x_2,x_3) rigidly attached to the body.

```
informal_definition coordinate_system where
  tag := "LL31.3"
  deps := [``RigidBody]
```

informal_lemma rigid_body_dof [source]

A rigid body in three-dimensional space has six degrees of freedom: three translational (for the position of its centre of mass) and three rotational (for its orientation).

```
informal_lemma rigid_body_dof where
  tag := "LL32-6DF"
  deps := [``RigidBody]
```

informal_lemma velocity_decomposition [source]

The velocity v of any point in a rigid body is $v = V + \Omega \times r$, where V is the velocity of the origin of the moving system and Ω is the angular velocity.

```
informal_lemma velocity_decomposition where
  tag := "LL31.L3"
  deps := [``RigidBody]
```

informal_lemma angular_velocity_is_well_defined [source]

The angular velocity of rotation of a rigid body from a system of coordinates fixed in the body is independent of the system chosen.

```
informal_lemma angular_velocity_is_well_defined where
  tag := "LL32-AM"
  deps := [``RigidBody]
```

informal_lemma decomposition_of_motion [source]

The motion of a rigid body can be decomposed into a translation of some reference point plus a rotation about that point. There exists a time-dependent vector $V(t)$ and angular velocity $\omega(t)$ such that $v(r) = V + \omega \times r$, where r is measured from the reference point.

```
informal_lemma decomposition_of_motion where
  tag := "LL32-DM"
  deps := [``RigidBody]
```

informal_lemma center_of_mass_point_moves_as_particle [source]

The centre of mass of a rigid body moves as if all mass were concentrated at that point and acted upon by the resultant external force: $M\ a_CM = \sum F_ext$.

```
informal_lemma center_of_mass_point_moves_as_particle where
  tag := "LL32-CM"
  deps := [``RigidBody]
```

informal_lemma angular_momentum_about_point [source]

The total angular momentum about a point O is $L = \int r \times v\ dm$. With $v = V + \omega \times r$ about the centre of mass, $L = R \times (M\ V) + I_CM\ \omega$, where R is the centre of mass position.

```
informal_lemma angular_momentum_about_point where
  tag := "LL32-L"
  deps := [``RigidBody]
```

informal_lemma translational_equation_inertial [source]

In the inertial frame, the translational equation of motion of a rigid body is given by $dP/dt = F$, where `P` is the total linear momentum and `F` is the total external force acting on the body.

```
informal_lemma translational_equation_inertial where
  tag := "LL32-TR"
  deps := [``RigidBody]
```

informal_lemma rotational_equation_inertial [source]

In the inertial frame, the rotational equation of motion of a rigid body about the center of mass is given by $dM/dt = K$, where `M` is the total angular momentum and `K` is the total external torque.

```
informal_lemma rotational_equation_inertial where
  tag := "LL32-ROT"
  deps := [``RigidBody]
```

informal_lemma kinetic_energy_decomposition [source]

The kinetic energy decomposes into translational and rotational parts: $T = (1/2)\ M\ |V|^2 + (1/2)\ \omega \cdot I_CM\ \omega$. Here V is the velocity of the centre of mass and I_CM is the inertia tensor about that point.

```
informal_lemma kinetic_energy_decomposition where
  tag := "LL32-TK"
  deps := [``RigidBody]
```

informal_lemma parallel_axis_theorem [source]

If I_O is the inertia tensor about a point O, then the inertia tensor about a parallel point O' displaced by a is I_{O'} = I_O + M(|a|² 1 − a ⊗ a). This is the parallel-axis theorem.

```
informal_lemma parallel_axis_theorem where
  tag := "LL32-PA"
  deps := [``RigidBody]
```

informal_definition principal_axes_of_inertia [source]

Because the inertia tensor is real symmetric, there exists an orthonormal basis of principal axes in which it is diagonal. The corresponding directions are the principal axes of inertia.

```
informal_definition principal_axes_of_inertia where
  tag := "LL32-PAE"
  deps := [``RigidBody]
```

informal_lemma principal_axes_of_inertia_bound [source]

None of the principal moments of inertia can exceed the sum of other two.

```
informal_lemma principal_axes_of_inertia_bound where
  tag := "LL32-PAEB"
  deps := [``RigidBody]
```

informal_definition asymmetrical_top [source]

An asymmetrical top is when none of the principal moments of inertia are equal.

```
informal_definition asymmetrical_top where
  tag := "LL32-AST"
  deps := [``RigidBody]
```

informal_definition symmetrical_top [source]

A symmetrical top is when only two of the principal moments of inertia are equal.

```
informal_definition symmetrical_top where
  tag := "LL32-ST"
  deps := [``RigidBody]
```

informal_definition spherical_top [source]

A spherical top is when all three of the principal moments of inertia are equal.

```
informal_definition spherical_top where
  tag := "LL32-SPT"
  deps := [``RigidBody]
```

informal_definition RotatingFrame [source]

A rotating body-fixed frame is a coordinate system attached to the body that rotates with the body relative to an inertial (fixed) frame. The frame is characterised by its angular velocity vector $\Omega(t)$.

```
informal_definition RotatingFrame where
  tag := "LL32-RF"
  deps := [``RigidBody]
```

informal_definition rotating_frame_derivative [source]

The time derivative in the rotating frame, d'/dt, is the derivative of the components of a vector with respect to time when expressed in the rotating (body-fixed) frame.

```
informal_definition rotating_frame_derivative where
  tag := "LL32-dprime"
  deps := [``RigidBody]
```

informal_lemma transport_law_inertial_rotating [source]

For any vector field $A(t)$, its inertial-frame time derivative equals the rotating-frame derivative plus the contribution from the frame rotation: $(dA/dt)_inertial = (dA/dt)_rotating + \Omega \times A$. Here Ω is the angular velocity of the rotating frame.

```
informal_lemma transport_law_inertial_rotating where
  tag := "LL36-Ltransport"
  deps := [``RigidBody]
```

informal_lemma transport_law_for_momentum [source]

For linear momentum, the relation between inertial and rotating derivatives is $(dP/dt)_inertial = d'P/dt + \Omega \times P$. So, $d'P/dt + \Omega \times P = F$ which is the linear-momentum equation in the rotating frame.

```
informal_lemma transport_law_for_momentum where
  tag := "LL32-transportP"
  deps := [``RigidBody]
```

informal_lemma transport_law_for_angular_momentum [source]

For angular momentum, the relation between inertial and rotating derivatives is $(dM/dt)_inertial = d'M/dt + \Omega \times M$, and with the rotational form of Newton's law $M_tot = (dM/dt)_inertial$ this yields $d'M/dt + \Omega \times M = K$, the angular-momentum equation in the rotating frame.

```
informal_lemma transport_law_for_angular_momentum where
  tag := "LL32-transportM"
  deps := [``RigidBody]
```

informal_lemma euler_equations [source]

When motion is described in body-fixed principal axes (I_1, I_2, I_3 diagonal), the equations of rotational motion (Euler's equations) are: $I_1\, d\omega_1/dt + (I_3 - I_2)\, \omega_2\, \omega_3 = M_1$, with cyclic permutations. M is the external torque about the centre of mass.

```
informal_lemma euler_equations where
  tag := "LL32-EQ"
  deps := [``RigidBody]
```

informal_lemma steady_rotation_conditions [source]

A rigid body can perform steady (uniform) rotation about any principal axis if the torque about that axis vanishes. Stability depends on the ordering of principal moments.

```
informal_lemma steady_rotation_conditions where
  tag := "LL32-SR"
  deps := [``RigidBody]
```

informal_lemma intermediate_axis_instability [source]

Rotations about the largest and smallest principal axes are stable under small perturbations; rotation about the intermediate axis is unstable (tennis-racket effect).

```
informal_lemma intermediate_axis_instability where
  tag := "LL32-IAI"
  deps := [``RigidBody]
```

informal_lemma reduction_to_two_body [source]

If a rigid body is confined to planar motion, its dynamics reduce to a two-dimensional problem: the inertia reduces to a scalar moment and rotation is described by a single angular velocity.

```
informal_lemma reduction_to_two_body where
  tag := "LL32-RTB"
  deps := [``RigidBody]
```

informal_lemma rigid_body_work_and_power [source]

The power delivered to a rigid body by forces is $P = \sum F_i \cdot v_i = F_tot \cdot V + M \cdot \omega$, where F_tot is total force, V the reference point velocity, and M the torque. Translational and rotational contributions separate.

```
informal_lemma rigid_body_work_and_power where
  tag := "LL32-WP"
  deps := [``RigidBody]
```

informal_lemma small_oscillations_about_equilibrium [source]

Small oscillations about a stable equilibrium orientation are governed by linearised equations obtained by expanding energy to second order in angular displacements. Normal modes and frequencies depend on inertia and restoring torques.

```
informal_lemma small_oscillations_about_equilibrium where
  tag := "LL32-SO"
  deps := [``RigidBody]
```

2.3 PhysLean.ClassicalMechanics.VectorFields

PhysLean/ClassicalMechanics/VectorFields.lean on GitHub.

lemma space_deriv_differentiable_time [source]

```
@[fun_prop]
lemma space_deriv_differentiable_time {d i} {M} [NormedAddCommGroup M] [NormedSpace ℝ M]
    {f : Time → Space d → M} (hf : ContDiff ℝ 2 ↿f) (x : Space d) :
    Differentiable ℝ (fun t => Space.deriv i (f t) x)
```

lemma time_deriv_differentiable_space [source]

```
@[fun_prop]
lemma time_deriv_differentiable_space {d } {M} [NormedAddCommGroup M] [NormedSpace ℝ M]
    {f : Time → Space d → M} (hf : ContDiff ℝ 2 ↿f) (t : Time) :
    Differentiable ℝ (fun x => Time.deriv (f · x) t)
```

lemma time_deriv_comm_space_deriv [source]

```
lemma time_deriv_comm_space_deriv {d i} {M} [NormedAddCommGroup M] [NormedSpace ℝ M]
    {f : Time → Space d → M} (hf : ContDiff ℝ 2 ↿f) (t : Time) (x : Space d) :
    Time.deriv (fun t' => Space.deriv i (f t') x) t
    = Space.deriv i (fun x' => Time.deriv (fun t' => f t' x') t) x
```

2.4 PhysLean.ClassicalMechanics.WaveEquation.Basic

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

lemma planeWave_eq [source]

```
lemma planeWave_eq {d f₀ c x} {s : Direction d} :
    planeWave f₀ c s t x = f₀ (⟪x, s.unit⟫_ℝ - c * t)
```

lemma planeWave_differentiable_time [source]

```
@[fun_prop]
lemma planeWave_differentiable_time {d f₀ c x} {s : Direction d}
    (h' : Differentiable ℝ f₀) :
    Differentiable ℝ (fun t => planeWave f₀ c s t x)
```

lemma planeWave_differentiable_space [source]

```
@[fun_prop]
lemma planeWave_differentiable_space {d f₀ c t} {s : Direction d}
    (h' : Differentiable ℝ f₀) :
    Differentiable ℝ (fun x => planeWave f₀ c s t x)
```

lemma planeWave_differentiable [source]

```
@[fun_prop]
lemma planeWave_differentiable {s : Direction d}
    {f₀ : ℝ → EuclideanSpace ℝ (Fin d)}
    (h' : Differentiable ℝ f₀) : Differentiable ℝ ↿(planeWave f₀ c s)
```

lemma planeWave_time_deriv [source]

```
lemma planeWave_time_deriv {d f₀ c x} {s : Direction d}
    (h' : Differentiable ℝ f₀) :
    ∂ₜ (planeWave f₀ c s · x) = -c • fun t => planeWave (fderiv ℝ f₀ · 1) c s t x
```

lemma planeWave_time_deriv_time_deriv [source]

```
lemma planeWave_time_deriv_time_deriv {d f₀ c x} {s : Direction d}
    (h' : ContDiff ℝ 2 f₀) :
    ∂ₜ (∂ₜ (planeWave f₀ c s · x)) = c ^ 2 • fun t => planeWave (iteratedDeriv 2 f₀) c s t x
```

lemma planeWave_space_deriv [source]

```
lemma planeWave_space_deriv {d f₀ c} {s : Direction d}
    (h' : Differentiable ℝ f₀) (i : Fin d) :
    Space.deriv i (planeWave f₀ c s t) =
    s.unit i • fun x => planeWave (fderiv ℝ f₀ · 1) c s t x
```

lemma planeWave_apply_space_deriv [source]

```
lemma planeWave_apply_space_deriv {d f₀ c} {s : Direction d}
    (h' : Differentiable ℝ f₀) (i j : Fin d) :
    Space.deriv i (fun x => planeWave f₀ c s t x j) =
    s.unit i • fun x => planeWave (fderiv ℝ f₀ · 1) c s t x j
```

lemma planeWave_space_deriv_space_deriv [source]

```
lemma planeWave_space_deriv_space_deriv {d f₀ c} {s : Direction d}
    (h' : ContDiff ℝ 2 f₀) (i : Fin d) :
    Space.deriv i (Space.deriv i (planeWave f₀ c s t)) =
    s.unit i ^ 2 • fun x => planeWave (iteratedDeriv 2 f₀ ·) c s t x
```

lemma planeWave_apply_space_deriv_space_deriv [source]

```
lemma planeWave_apply_space_deriv_space_deriv {d f₀ c} {s : Direction d}
    (h' : ContDiff ℝ 2 f₀) (i j : Fin d) :
    Space.deriv i (Space.deriv i (fun x => planeWave f₀ c s t x j)) =
    (s.unit i) ^ 2 •fun x => planeWave (iteratedDeriv 2 f₀ ·) c s t x j
```

lemma planeWave_laplacian [source]

```
lemma planeWave_laplacian {d f₀ c} {s : Direction d} (h' : ContDiff ℝ 2 f₀) :
    Δ (planeWave f₀ c s t) = fun x => planeWave (iteratedDeriv 2 f₀ ·) c s t x
```

theorem planeWave_waveEquation [source]

The plane wave satisfies the wave equation.

```
theorem planeWave_waveEquation (c : ℝ) (s : Direction d)
    (f₀ : ℝ → EuclideanSpace ℝ (Fin d)) (hf₀ : ContDiff ℝ 2 f₀) :
    ∀ t x, WaveEquation (planeWave f₀ c s) t x c
```

2.5 PhysLean.Cosmology.FLRW.Basic

PhysLean/Cosmology/FLRW/Basic.lean on GitHub.

lemma mul_sinh_as_div [source]

The limit of `S (Saddle k) r` as `k → ∞` is equal to `S (Flat) r`. First show that `k * sinh(r / k) = sinh(r / k) / (1 / k)` pointwise.

```
lemma mul_sinh_as_div (r k : ℝ) :
    k * Real.sinh (r / k) = Real.sinh (r / k) / (1 / k)
```

lemma tendsto_sinh_rx_over_x [source]

First, show that limit of `sinh(r * x) / x` is r at the limit x goes to zero. Then the next theorem will address the rewrite using Filter.Tendsto.comp

```
@[sorryful]
lemma tendsto_sinh_rx_over_x (r : ℝ) :
    Tendsto (fun x : ℝ => Real.sinh (r * x) / x) (𝓝[≠] 0) (𝓝 r)
```

lemma mul_sin_as_div [source]

The limit of `S (Sphere k) r` as `k → ∞` is equal to `S (Flat) r`. First show that `k * sinh(r / k) = sin(r / k) / (1 / k)` pointwise.

```
lemma mul_sin_as_div (r k : ℝ) :
    k * Real.sin (r / k) = Real.sin (r / k) / (1 / k)
```

lemma tendsto_sin_rx_over_x [source]

First, show that limit of `sin(r * x) / x` is r at the limit x goes to zero. Then the next theorem will address the rewrite using Filter.Tendsto.comp

```
@[sorryful]
lemma tendsto_sin_rx_over_x (r : ℝ) :
    Tendsto (fun x : ℝ => Real.sin (r * x) / x) (𝓝[≠] 0) (𝓝 r)
```

def FirstOrderFriedmann [source]

The first-order Friedmann equation.

- `a : ℝ → ℝ` is the FLRW scale factor as a function of cosmic time `t`. - `ρ : ℝ → ℝ` is the total energy density as a function of cosmic time `t`. - `k : ℝ` is the spatial curvature parameter. - `Λ : ℝ` is the cosmological constant. - `G : ℝ` is Newton's constant. - `c : ℝ` is the speed of light. It may be set to 1 for convenience.

Note: We will leave `c` explicit for generality and accounting purposes.

At time `t` the equation reads: `(a'(t) / a(t))^2 = (8πG/3) ρ(t) - k c^2 / a(t)^2 + Λ c^2 / 3`.

```
def FirstOrderFriedmann (a ρ: ℝ → ℝ) (k Λ G c : ℝ) (t : ℝ) : Prop
```

def SecondOrderFriedmann [source]

The second-order Friedmann equation. Note: Other sources may call this the Raychaudhuri equation. We choose not to use that terminology to avoid the Raychaudhuri equation related to describing congruences of geodesics in general relativity. - `a : ℝ → ℝ` is the FLRW scale factor as a function of cosmic time `t`. - `ρ : ℝ → ℝ` is the total energy density as a function of cosmic time `t`. - `p : ℝ → ℝ` is the pressure. It is related to `ρ` via `p = w * ρ` - `w` is the equation of state. We will introduce this later. - `Λ : ℝ` is the cosmological constant. - `G : ℝ` is Newton's constant. - `c : ℝ` is the speed of light. It may be set to 1 for convenience.

Note: We will leave `c` explicit for generality and accounting purposes.

At time `t` the equation reads: `(a''(t) / a (t)) = - (4πG/3) * (ρ(t) + 3 * p(t) / c^2) + Λ * c^2 / 3`.

```
def SecondOrderFriedmann (a ρ p: ℝ → ℝ) (Λ G c : ℝ) (t : ℝ) : Prop
```

def hubbleConstant [source]

The hubble constant defined in terms of the scale factor as `(dₜ a) / a`.

The notation `H` is used for the `hubbleConstant`.

Semiformal implementation note: Implement also scoped notation.

```
noncomputable def hubbleConstant (a : ℝ → ℝ) (t : ℝ) : ℝ
```

def decelerationParameter [source]

The deceleration parameter defined in terms of the scale factor as `- (dₜdₜ a) a / (dₜ a)^2`.

The notation `q` is used for the `decelerationParameter`.

Semiformal implementation note: Implement also scoped notation.

```
noncomputable def decelerationParameter (a : ℝ → ℝ) (t : ℝ) : ℝ
```

informal_lemma decelerationParameter_eq_one_plus_hubbleConstant [source]

The deceleration parameter is equal to `- (1 + (dₜ H)/H^2)`.

```
informal_lemma decelerationParameter_eq_one_plus_hubbleConstant where
  deps := []
  tag := "6Z23H"
```

informal_lemma time_evolution_hubbleConstant [source]

The time evolution of the hubble parameter is equal to `dₜ H = - H^2 (1 + q)`.

```
informal_lemma time_evolution_hubbleConstant where
  deps := []
  tag := "6Z3BS"
```

informal_lemma hubbleConstant_decrease_iff [source]

There exists a time at which the hubble constant decreases if and only if there exists a time where the deceleration parameter is less then `-1`.

```
informal_lemma hubbleConstant_decrease_iff where
  deps := []
  tag := "6Z3FS"
```

2.6 PhysLean.Electromagnetism.Distributions.Potential

PhysLean/Electromagnetism/Distributions/Potential.lean on GitHub.

abbrev ElectromagneticPotentialD [source]

The electromagnetic potential is a tensor `A^μ`.

```
noncomputable abbrev ElectromagneticPotentialD (d : ℕ := 3)
```

def ElectromagneticPotentialD.toComponents [source]

The linear map from an electromagnetic potential to its components.

```
noncomputable def toComponents {d : ℕ} :
    ElectromagneticPotentialD d ≃ₗ[ℝ] ((Fin 1 ⊕ Fin d) → (SpaceTime d) →d[ℝ] ℝ) where
```

def fieldStrengthMatrix [source]

The field strength matrix with indices `F^μ^ν`.

```
noncomputable def fieldStrengthMatrix {d : ℕ} :
    ElectromagneticPotentialD d →ₗ[ℝ]
    ((Fin 1 ⊕ Fin d) × (Fin 1 ⊕ Fin d) → (SpaceTime d) →d[ℝ] ℝ) where
```

lemma fieldStrengthMatrix_same_same [source]

```
@[simp]
lemma fieldStrengthMatrix_same_same {d : ℕ} (A : ElectromagneticPotentialD d) (μ : Fin 1 ⊕ Fin
    d) :
    A.fieldStrengthMatrix (μ, μ) = 0
```

lemma fieldStrengthMatrix_antisymm [source]

```
lemma fieldStrengthMatrix_antisymm {d : ℕ} (A : ElectromagneticPotentialD d)
    (μ ν : Fin 1 ⊕ Fin d) :
    A.fieldStrengthMatrix (μ, ν) = - A.fieldStrengthMatrix (ν, μ)
```

def scalarPotential [source]

The scalar potential from an electromagnetic potential which is a distribution.

```
noncomputable def scalarPotential {d} :
    ElectromagneticPotentialD d →ₗ[ℝ] (Time × Space d) →d[ℝ] ℝ where
```

def vectorPotential [source]

The vector potential from an electromagnetic potential which is a distribution.

```
noncomputable def vectorPotential {d}:
    ElectromagneticPotentialD d →ₗ[ℝ] (Time × Space d) →d[ℝ] EuclideanSpace ℝ (Fin d) where
```

def toComponentsEuclidean [source]

The linear map taking a distribution on Euclidean space to its components.

```
noncomputable def toComponentsEuclidean {d : ℕ} :
    ((Time × Space d) →d[ℝ] EuclideanSpace ℝ (Fin d)) ≃ₗ[ℝ]
    (Fin d → (Time × Space d) →d[ℝ] ℝ) where
```

lemma toComponentsEuclidean_apply [source]

```
@[simp]
lemma toComponentsEuclidean_apply {d : ℕ} (E : (Time × Space d) →d[ℝ] EuclideanSpace ℝ (Fin d))
    (i : Fin d) (ε : 𝓢(Time × Space d, ℝ)) :
    (toComponentsEuclidean E i) ε = E ε i
```

def electricField [source]

The electric field associated with a electromagnetic potential which is a distribution.

```
noncomputable def electricField {d} :
    ElectromagneticPotentialD d →ₗ[ℝ] (Time × Space d) →d[ℝ] EuclideanSpace ℝ (Fin d) where
```

lemma electricField_fieldStrengthMatrix [source]

```
lemma electricField_fieldStrengthMatrix {d} {A : ElectromagneticPotentialD d} (i : Fin d) :
    toComponentsEuclidean A.electricField i =
    timeSliceD (A.fieldStrengthMatrix (Sum.inr i, Sum.inl 0))
```

lemma fieldStrengthMatrix_col_eq_electricField [source]

```
lemma fieldStrengthMatrix_col_eq_electricField {d} {A : ElectromagneticPotentialD d}
    (i : Fin d) :
    (A.fieldStrengthMatrix (Sum.inr i, Sum.inl 0)) =
    timeSliceD.symm (toComponentsEuclidean A.electricField i)
```

lemma fieldStrengthMatrix_row_eq_electricField [source]

```
lemma fieldStrengthMatrix_row_eq_electricField {d} {A : ElectromagneticPotentialD d}
    (i : Fin d) :
    (A.fieldStrengthMatrix (Sum.inl 0, Sum.inr i)) =
    - timeSliceD.symm (toComponentsEuclidean A.electricField i)
```

def magneticField [source]

The magnetic field associated with a electromagnetic potential in 3 dimensions.

```
noncomputable def magneticField :
    ElectromagneticPotentialD 3 →ₗ[ℝ] (Time × Space 3) →d[ℝ] EuclideanSpace ℝ (Fin 3) where
```

abbrev LorentzCurrentDensityD [source]

The Lorentz current density (aka four-current) as a distribution.

```
abbrev LorentzCurrentDensityD (d : ℕ := 3)
```

def LorentzCurrentDensityD.toComponents [source]

The linear map taking a Lorentz current density to its components.

```
noncomputable def toComponents {d : ℕ} :
    LorentzCurrentDensityD d ≃ₗ[ℝ] ((Fin 1 ⊕ Fin d) → (SpaceTime d) →d[ℝ] ℝ) where
```

def gradLagrangian [source]

The variational gradient of the lagrangian for an electromagnetic potential which is a distribution. This is defined nor proved for distributions.

```
noncomputable def gradLagrangian {d : ℕ} (A : ElectromagneticPotentialD d)
    (J : LorentzCurrentDensityD d) :
    (Fin 1 ⊕ Fin d) → (SpaceTime d) →d[ℝ] ℝ
```

lemma gradLagrangian_one_dimension_electricField [source]

```
lemma gradLagrangian_one_dimension_electricField (A : ElectromagneticPotentialD 1)
    (J : LorentzCurrentDensityD 1) :
    A.gradLagrangian J = fun μ =>
      match μ with
      | Sum.inl 0 => SpaceTime.timeSliceD.symm
          (Space.spaceDivD A.electricField) - J.toComponents (Sum.inl 0)
      | Sum.inr 0 => J.toComponents (Sum.inr 0) +
        SpaceTime.timeSliceD.symm
        (toComponentsEuclidean (Space.timeDerivD A.electricField) 0)
```

2.7 PhysLean.Electromagnetism.Dynamics.Basic

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

structure FreeSpace [source]

Free space consists of the specification of the electric permittivity and the magnetic permeability.

```
structure FreeSpace where
  ε₀ : ℝ
  μ₀ : ℝ
  ε₀_pos : 0 < ε₀
  μ₀_pos : 0 < μ₀
```

lemma ε₀_nonneg [source]

```
@[simp]
lemma ε₀_nonneg : 0 ≤ 𝓕.ε₀
```

lemma μ₀_nonneg [source]

```
@[simp]
lemma μ₀_nonneg : 0 ≤ 𝓕.μ₀
```

lemma ε₀_ne_zero [source]

```
@[simp]
lemma ε₀_ne_zero : 𝓕.ε₀ ≠ 0
```

lemma μ₀_ne_zero [source]

```
@[simp]
lemma μ₀_ne_zero : 𝓕.μ₀ ≠ 0
```

def c [source]

The speed of light in free space.

```
noncomputable def c : SpeedOfLight
```

lemma c_val [source]

```
lemma c_val : (𝓕.c : ℝ) = 1 / √(𝓕.ε₀ * 𝓕.μ₀)
```

lemma c_sq [source]

```
lemma c_sq : (𝓕.c : ℝ) ^ 2 = 1 / (𝓕.ε₀ * 𝓕.μ₀)
```

lemma c_abs [source]

```
@[simp]
lemma c_abs : abs (𝓕.c : ℝ) = 𝓕.c
```

2.8 PhysLean.Electromagnetism.Dynamics.CurrentDensity

PhysLean/Electromagnetism/Dynamics/CurrentDensity.lean on GitHub.

abbrev LorentzCurrentDensity [source]

The Lorentz current density, also called four-current.

```
abbrev LorentzCurrentDensity (d : ℕ := 3)
```

def chargeDensity [source]

The underlying charge density associated with a Lorentz current density.

```
noncomputable def chargeDensity (c : SpeedOfLight := 1)
    (J : LorentzCurrentDensity d) : Time → Space d → ℝ
```

lemma chargeDensity_eq_timeSlice [source]

```
lemma chargeDensity_eq_timeSlice {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity d} :
    J.chargeDensity c = timeSlice c (fun x => (1 / (c : ℝ)) • J x (Sum.inl 0))
```

lemma chargeDensity_zero [source]

```
@[simp]
lemma chargeDensity_zero {d : ℕ} {c : SpeedOfLight}:
    chargeDensity c (0 : LorentzCurrentDensity d) = 0
```

lemma chargeDensity_differentiable [source]

```
lemma chargeDensity_differentiable {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity d}
    (hJ : Differentiable ℝ J) : Differentiable ℝ ↿(J.chargeDensity c)
```

lemma chargeDensity_differentiable_space [source]

```
lemma chargeDensity_differentiable_space {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity
    d}
    (hJ : Differentiable ℝ J) (t : Time) :
    Differentiable ℝ (fun x => J.chargeDensity c t x)
```

lemma chargeDensity_contDiff [source]

```
lemma chargeDensity_contDiff {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity d}
    (hJ : ContDiff ℝ n J) : ContDiff ℝ n ↿(J.chargeDensity c)
```

def currentDensity [source]

The underlying (non-Lorentz) current density associated with a Lorentz current density.

```
noncomputable def currentDensity (c : SpeedOfLight := 1) (J : LorentzCurrentDensity d) :
    Time → Space d → EuclideanSpace ℝ (Fin d)
```

lemma currentDensity_eq_timeSlice [source]

```
lemma currentDensity_eq_timeSlice {d : ℕ} {J : LorentzCurrentDensity d} :
    J.currentDensity c = timeSlice c (fun x i => J x (Sum.inr i))
```

lemma currentDensity_zero [source]

```
@[simp]
lemma currentDensity_zero {d : ℕ} {c : SpeedOfLight}:
    currentDensity c (0 : LorentzCurrentDensity d) = 0
```

lemma currentDensity_differentiable [source]

```
lemma currentDensity_differentiable {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity d}
    (hJ : Differentiable ℝ J) : Differentiable ℝ ↿(J.currentDensity c)
```

lemma currentDensity_apply_differentiable [source]

```
lemma currentDensity_apply_differentiable {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity
     d}
    (hJ : Differentiable ℝ J) (i : Fin d) :
    Differentiable ℝ ↿(fun t x => J.currentDensity c t x i)
```

lemma currentDensity_differentiable_space [source]

```
lemma currentDensity_differentiable_space {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity
     d}
    (hJ : Differentiable ℝ J) (t : Time) :
    Differentiable ℝ (fun x => J.currentDensity c t x)
```

lemma currentDensity_apply_differentiable_space [source]

```
lemma currentDensity_apply_differentiable_space {d : ℕ} {c : SpeedOfLight}
    {J : LorentzCurrentDensity d}
    (hJ : Differentiable ℝ J) (t : Time) (i : Fin d) :
    Differentiable ℝ (fun x => J.currentDensity c t x i)
```

lemma currentDensity_differentiable_time [source]

```
lemma currentDensity_differentiable_time {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity
    d}
    (hJ : Differentiable ℝ J) (x : Space d) :
    Differentiable ℝ (fun t => J.currentDensity c t x)
```

lemma currentDensity_apply_differentiable_time [source]

```
lemma currentDensity_apply_differentiable_time {d : ℕ} {c : SpeedOfLight}
    {J : LorentzCurrentDensity d}
    (hJ : Differentiable ℝ J) (x : Space d) (i : Fin d) :
    Differentiable ℝ (fun t => J.currentDensity c t x i)
```

lemma currentDensity_ContDiff [source]

```
lemma currentDensity_ContDiff {d : ℕ} {c : SpeedOfLight} {J : LorentzCurrentDensity d}
    (hJ : ContDiff ℝ n J) : ContDiff ℝ n ↿(J.currentDensity c)
```

2.9 PhysLean.Electromagnetism.Dynamics.Hamiltonian

PhysLean/Electromagnetism/Dynamics/Hamiltonian.lean on GitHub.

def canonicalMomentum [source]

The canonical momentum associated with the lagrangian of an electromagnetic potential and a Lorentz current density.

```
noncomputable def canonicalMomentum (𝓕 : FreeSpace) (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) :
    SpaceTime d → Lorentz.Vector d
```

lemma canonicalMomentum_eq_gradient_kineticTerm [source]

```
lemma canonicalMomentum_eq_gradient_kineticTerm {d}
    {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 2 A) (J : LorentzCurrentDensity d) :
    A.canonicalMomentum 𝓕 J = fun x =>
    gradient (fun (v : Lorentz.Vector d) =>
    kineticTerm 𝓕 (fun x => A x + x (Sum.inl 0) • v) x) 0
```

lemma canonicalMomentum_eq [source]

```
lemma canonicalMomentum_eq {d} {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 2 A) (J : LorentzCurrentDensity d) :
    A.canonicalMomentum 𝓕 J = fun x => fun μ =>
      (1/𝓕.μ₀) * η μ μ • A.fieldStrengthMatrix x (μ, Sum.inl 0)
```

lemma canonicalMomentum_eq_electricField [source]

```
lemma canonicalMomentum_eq_electricField {d} {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 2 A) (J : LorentzCurrentDensity d) :
    A.canonicalMomentum 𝓕 J = fun x => fun μ =>
      match μ with
      | Sum.inl 0 => 0
      | Sum.inr i => - (1/(𝓕.μ₀ * 𝓕.c)) * A.electricField 𝓕.c (x.time 𝓕.c) x.space i
```

def hamiltonian [source]

The Hamiltonian associated with an electromagnetic potential and a Lorentz current density.

```
noncomputable def hamiltonian (𝓕 : FreeSpace) (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) (x : SpaceTime d) : ℝ
```

lemma hamiltonian_eq_electricField_vectorPotential [source]

```
lemma hamiltonian_eq_electricField_vectorPotential {d} {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (hA : ContDiff ℝ 2 A)
    (J : LorentzCurrentDensity d) (x : SpaceTime d) :
    A.hamiltonian 𝓕 J x =
      - (1/ 𝓕.c.val^2 * 𝓕.μ₀⁻¹) * ∑ i, A.electricField 𝓕.c (x.time 𝓕.c) x.space i *
      (∂ₜ (A.vectorPotential 𝓕.c · x.space) (x.time 𝓕.c) i) - lagrangian 𝓕 A J x
```

lemma hamiltonian_eq_electricField_scalarPotential [source]

```
lemma hamiltonian_eq_electricField_scalarPotential {d} {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (hA : ContDiff ℝ 2 A)
    (J : LorentzCurrentDensity d) (x : SpaceTime d) :
    A.hamiltonian 𝓕 J x =
      (1/ 𝓕.c.val^2 * 𝓕.μ₀⁻¹) * (‖A.electricField 𝓕.c (x.time 𝓕.c) x.space‖ ^ 2
```

```
    + ⟪A.electricField 𝓕.c (x.time 𝓕.c) x.space,
      Space.grad (A.scalarPotential 𝓕.c (x.time 𝓕.c) ·) x.space⟫_ℝ)
      - lagrangian 𝓕 A J x
```

lemma hamiltonian_eq_electricField_magneticField [source]

```
lemma hamiltonian_eq_electricField_magneticField (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 2 A) (J : LorentzCurrentDensity d) (x : SpaceTime d) :
    A.hamiltonian 𝓕 J x = 1/2 * 𝓕.ε₀ * (‖A.electricField 𝓕.c (x.time 𝓕.c) x.space‖ ^ 2
      + 𝓕.c ^ 2 / 2 * ∑ i, ∑ j, ‖A.magneticFieldMatrix 𝓕.c (x.time 𝓕.c) x.space (i, j)‖ ^ 2)
      + 𝓕.ε₀ * ⟪A.electricField 𝓕.c (x.time 𝓕.c) x.space,
        Space.grad (A.scalarPotential 𝓕.c (x.time 𝓕.c) ·) x.space⟫_ℝ
      + A.scalarPotential 𝓕.c (x.time 𝓕.c) x.space * J.chargeDensity 𝓕.c (x.time 𝓕.c) x.
    space
      - ∑ i, A.vectorPotential 𝓕.c (x.time 𝓕.c) x.space i *
        J.currentDensity 𝓕.c (x.time 𝓕.c) x.space i
```

2.10 PhysLean.Electromagnetism.Dynamics.IsExtrema

PhysLean/Electromagnetism/Dynamics/IsExtrema.lean on GitHub.

def IsExtrema [source]

The condition on an electromagnetic potential to be an extrema of the lagrangian.

```
def IsExtrema {d} (𝓕 : FreeSpace) (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) : Prop
```

lemma isExtrema_iff_gradLagrangian [source]

```
lemma isExtrema_iff_gradLagrangian {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) :
    IsExtrema 𝓕 A J ↔ A.gradLagrangian 𝓕 J = 0
```

lemma isExtrema_iff_fieldStrengthMatrix [source]

```
lemma isExtrema_iff_fieldStrengthMatrix {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d) (hJ : ContDiff ℝ ∞ J) :
    IsExtrema 𝓕 A J ↔
    ∀ x, ∀ ν, ∑ μ, ∂_ μ (A.fieldStrengthMatrix · (μ, ν)) x = 𝓕.μ₀ * J x ν
```

lemma isExtrema_iff_gauss_ampere_magneticFieldMatrix [source]

```
lemma isExtrema_iff_gauss_ampere_magneticFieldMatrix {d} {𝓕 : FreeSpace}
    {A : ElectromagneticPotential d}
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) :
    IsExtrema 𝓕 A J ↔ ∀ t, ∀ x, (∇ · (A.electricField 𝓕.c t)) x = J.chargeDensity 𝓕.c t x /
    𝓕.ε₀
    ∧ ∀ i, 𝓕.μ₀ * 𝓕.ε₀ * ∂ₜ (fun t => A.electricField 𝓕.c t x) t i =
    ∑ j, ∂[j] (A.magneticFieldMatrix 𝓕.c t · (j, i)) x - 𝓕.μ₀ * J.currentDensity 𝓕.c t x i
```

lemma time_deriv_electricField_of_isExtrema [source]

```
lemma time_deriv_electricField_of_isExtrema {A : ElectromagneticPotential d}
    {𝓕 : FreeSpace}
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d) (hJ : ContDiff ℝ ∞ J)
    (h : IsExtrema 𝓕 A J) (t : Time) (x : Space d) (i : Fin d) :
    ∂ₜ (A.electricField 𝓕.c · x) t i =
      1 / (𝓕.μ₀ * 𝓕.ε₀) * ∑ j, ∂[j] (A.magneticFieldMatrix 𝓕.c t · (j, i)) x -
      (1/ 𝓕.ε₀) * J.currentDensity 𝓕.c t x i
```

lemma time_deriv_time_deriv_magneticFieldMatrix_of_isExtrema [source]

```
lemma time_deriv_time_deriv_magneticFieldMatrix_of_isExtrema {A : ElectromagneticPotential d}
    {𝓕 : FreeSpace}
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d) (hJd : Differentiable ℝ J)
    (hJ : ContDiff ℝ ∞ J) (h : IsExtrema 𝓕 A J)
    (t : Time) (x : Space d) (i j : Fin d) :
    ∂ₜ (∂ₜ (A.magneticFieldMatrix 𝓕.c · x (i, j))) t =
    𝓕.c ^ 2 * ∑ k, ∂[k] (∂[k] (A.magneticFieldMatrix 𝓕.c t · (i, j))) x +
    𝓕.ε₀⁻¹ * (∂[j] (J.currentDensity 𝓕.c t · i) x - ∂[i] (J.currentDensity 𝓕.c t · j) x)
```

lemma time_deriv_time_deriv_electricField_of_isExtrema [source]

```
lemma time_deriv_time_deriv_electricField_of_isExtrema {A : ElectromagneticPotential d}
    {𝓕 : FreeSpace}
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) (h : IsExtrema 𝓕 A J)
    (t : Time) (x : Space d) (i : Fin d) :
    ∂ₜ (∂ₜ (A.electricField 𝓕.c · x i)) t =
      𝓕.c ^ 2 * ∑ j, (∂[j] (∂[j] (A.electricField 𝓕.c t · i)) x) -
      𝓕.c ^ 2 / 𝓕.ε₀ * ∂[i] (J.chargeDensity 𝓕.c t ·) x -
      𝓕.c ^ 2 * 𝓕.μ₀ * ∂ₜ (J.currentDensity 𝓕.c · x i) t
```

2.11 PhysLean.Electromagnetism.Dynamics.KineticTerm

PhysLean/Electromagnetism/Dynamics/KineticTerm.lean on GitHub.

lemma kineticTerm_eq_sum_fieldStrengthMatrix [source]

```
lemma kineticTerm_eq_sum_fieldStrengthMatrix {d} {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (x : SpaceTime d) : A.kineticTerm 𝓕 x =
    - 1/(4 * 𝓕.μ₀) * ∑ μ, ∑ ν, ∑ μ', ∑ ν', η μ μ' * η ν ν' *
      A.fieldStrengthMatrix x (μ, ν) * A.fieldStrengthMatrix x (μ', ν')
```

lemma kineticTerm_eq_sum_fieldStrengthMatrix_sq [source]

```
lemma kineticTerm_eq_sum_fieldStrengthMatrix_sq {d} {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (x : SpaceTime d) : A.kineticTerm 𝓕 x =
    - 1/(4 * 𝓕.μ₀) * ∑ μ, ∑ ν, η μ μ * η ν ν * ‖A.fieldStrengthMatrix x (μ, ν)‖ ^ 2
```

lemma kineticTerm_eq_electric_magnetic' [source]

```
lemma kineticTerm_eq_electric_magnetic' {𝓕 : FreeSpace} {A : ElectromagneticPotential}
    (hA : Differentiable ℝ A) (x : SpaceTime) :
    A.kineticTerm 𝓕 x =
```

```
    1/2 * (𝓕.ε₀ * ‖A.electricField 𝓕.c (x.time 𝓕.c) x.space‖ ^ 2 -
      (1 / 𝓕.μ₀) * ‖A.magneticField 𝓕.c (x.time 𝓕.c) x.space‖ ^ 2)
```

lemma kineticTerm_eq_electricMatrix_magneticFieldMatrix_time_space [source]

```
lemma kineticTerm_eq_electricMatrix_magneticFieldMatrix_time_space {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (t : Time)
    (x : Space d) (hA : Differentiable ℝ A) :
    A.kineticTerm 𝓕 ((toTimeAndSpace 𝓕.c).symm (t, x)) =
    1/2 * (𝓕.ε₀ * ‖A.electricField 𝓕.c t x‖ ^ 2 -
    (1 / (2 * 𝓕.μ₀)) * ∑ i, ∑ j, ‖A.magneticFieldMatrix 𝓕.c t x (i, j)‖ ^ 2)
```

lemma kineticTerm_eq_electricMatrix_magneticFieldMatrix [source]

```
lemma kineticTerm_eq_electricMatrix_magneticFieldMatrix {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (x : SpaceTime d)
    (hA : Differentiable ℝ A) :
    A.kineticTerm 𝓕 x =
    1/2 * (𝓕.ε₀ * ‖A.electricField 𝓕.c (x.time 𝓕.c) x.space‖ ^ 2 -
    (1 / (2 * 𝓕.μ₀)) * ∑ i, ∑ j, ‖A.magneticFieldMatrix 𝓕.c (x.time 𝓕.c) x.space (i, j)‖ ^ 2)
```

lemma kineticTerm_const [source]

```
lemma kineticTerm_const {d} {𝓕 : FreeSpace} (A₀ : Lorentz.Vector d) :
    kineticTerm 𝓕 (fun _ : SpaceTime d => A₀) = 0
```

lemma kineticTerm_add_const [source]

```
lemma kineticTerm_add_const {d} {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (A₀ : Lorentz.Vector d) :
    kineticTerm 𝓕 (fun x => A x + A₀) = kineticTerm 𝓕 A
```

lemma kineticTerm_contDiff [source]

```
lemma kineticTerm_contDiff {d} {n : WithTop ℕ∞} {𝓕 : FreeSpace} (A : ElectromagneticPotential
    d)
    (hA : ContDiff ℝ (n + 1) A) :
    ContDiff ℝ n (A.kineticTerm 𝓕)
```

lemma kineticTerm_add_time_mul_const [source]

```
lemma kineticTerm_add_time_mul_const {d} {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (ha : Differentiable ℝ A)
    (c : Lorentz.Vector d) (x : SpaceTime d) :
    kineticTerm 𝓕 (fun x => A x + x (Sum.inl 0) • c) x = A.kineticTerm 𝓕 x +
        (-1 / (2 * 𝓕.μ₀) * ∑ ν, ((2 * c ν * η ν ν * ∂_ (Sum.inl 0) A x ν + η ν ν * c ν ^ 2 -
        2 * c ν * (∂_ ν A x (Sum.inl 0)))) + 1/(2 * 𝓕.μ₀) * c (Sum.inl 0) ^2)
```

instance InnerProductSpace ℝ (SpaceTime d) [source]

A local instance of an inner product structure on `SpaceTime`.

```
noncomputable local instance {d} : InnerProductSpace ℝ (SpaceTime d)
```

lemma gradKineticTerm_eq_electric_magnetic_three [source]

```
lemma gradKineticTerm_eq_electric_magnetic_three {𝓕 : FreeSpace} (A : ElectromagneticPotential
    )
    (x : SpaceTime) (ha : ContDiff ℝ ∞ A) :
    A.gradKineticTerm 𝓕 x =
    (1/(𝓕.μ₀ * 𝓕.c) * Space.div (A.electricField 𝓕.c (x.time 𝓕.c)) x.space) •
      Lorentz.Vector.basis (Sum.inl 0) +
    ∑ i, (𝓕.μ₀⁻¹ * (1 / 𝓕.c ^ 2 * ∂ₜ (fun t => A.electricField 𝓕.c t x.space) (x.time 𝓕.c) i-

      Space.curl (A.magneticField 𝓕.c (x.time 𝓕.c)) x.space i)) •
      Lorentz.Vector.basis (Sum.inr i)
```

lemma gradKineticTerm_add [source]

```
lemma gradKineticTerm_add {d} {𝓕 : FreeSpace} (A1 A2 : ElectromagneticPotential d)
    (hA1 : ContDiff ℝ ∞ A1) (hA2 : ContDiff ℝ ∞ A2) :
    (A1 + A2).gradKineticTerm 𝓕 = A1.gradKineticTerm 𝓕 + A2.gradKineticTerm 𝓕
```

lemma gradKineticTerm_smul [source]

```
lemma gradKineticTerm_smul {d} {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ ∞ A) (c : ℝ) :
    (c • A).gradKineticTerm 𝓕 = c • A.gradKineticTerm 𝓕
```

lemma kineticTerm_hasVarGradientAt [source]

```
lemma kineticTerm_hasVarGradientAt {d} {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ ∞ A) : HasVarGradientAt (kineticTerm 𝓕) (A.gradKineticTerm 𝓕) A
```

2.12 PhysLean.Electromagnetism.Dynamics.Lagrangian

PhysLean/Electromagnetism/Dynamics/Lagrangian.lean on GitHub.

def freeCurrentPotential [source]

The potential energy from the interaction of the electromagnetic potential with the free current `J`.

```
noncomputable def freeCurrentPotential (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d)
    (x : SpaceTime d) : ℝ
```

lemma freeCurrentPotential_add_const [source]

```
lemma freeCurrentPotential_add_const (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) (c : Lorentz.Vector d) (x : SpaceTime d) :
    freeCurrentPotential (fun x => A x + c) J x = freeCurrentPotential A J x + ⟪c, J x⟫ₘ
```

lemma freeCurrentPotential_hasVarGradientAt [source]

```
lemma freeCurrentPotential_hasVarGradientAt (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) :
```

```
    HasVarGradientAt (fun A => freeCurrentPotential A J)
    (((∑ μ, fun x => (η μ μ * J x μ) • Lorentz.Vector.basis μ))) A
```

lemma freeCurrentPotential_eq_sum_scalarPotential_vectorPotential [source]

```
lemma freeCurrentPotential_eq_sum_scalarPotential_vectorPotential
    (ℱ : FreeSpace) (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) (x : SpaceTime d) :
    A.freeCurrentPotential J x =
    A.scalarPotential ℱ.c (x.time ℱ.c) x.space * J.chargeDensity ℱ.c (x.time ℱ.c) x.space
    - ∑ i, A.vectorPotential ℱ.c (x.time ℱ.c) x.space i *
        J.currentDensity ℱ.c (x.time ℱ.c) x.space i
```

def gradFreeCurrentPotential [source]

The variational gradient of the free current potential.

```
noncomputable def gradFreeCurrentPotential {d} (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) : SpaceTime d → Lorentz.Vector d
```

lemma gradFreeCurrentPotential_eq_sum_basis [source]

```
lemma gradFreeCurrentPotential_eq_sum_basis {d} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) :
    A.gradFreeCurrentPotential J = (∑ μ, fun x => (η μ μ * J x μ) • Lorentz.Vector.basis μ)
```

lemma gradFreeCurrentPotential_eq_chargeDensity_currentDensity [source]

```
lemma gradFreeCurrentPotential_eq_chargeDensity_currentDensity {d}
    (ℱ : FreeSpace) (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) (x : SpaceTime d) :
    A.gradFreeCurrentPotential J x =
      (ℱ.c * J.chargeDensity ℱ.c (x.time ℱ.c) x.space) • Lorentz.Vector.basis (Sum.inl 0) +
      (∑ i, - J.currentDensity ℱ.c (x.time ℱ.c) x.space i • Lorentz.Vector.basis (Sum.inr i))
```

def lagrangian [source]

The lagrangian density associated with a electromagnetic potential and a Lorentz current density.

```
noncomputable def lagrangian (ℱ : FreeSpace) (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) (x : SpaceTime d) : ℝ
```

lemma lagrangian_add_const [source]

```
lemma lagrangian_add_const {d} {ℱ : FreeSpace} (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) (c : Lorentz.Vector d) (x : SpaceTime d) :
    lagrangian ℱ (fun x => A x + c) J x = lagrangian ℱ A J x - ⟪c, J x⟫ₘ
```

lemma lagrangian_eq_electric_magnetic [source]

The Lagrangian is equal to `1/2 * (ε₀ E^2 - 1/μ₀ B^2) - φρ + A · j`

```
lemma lagrangian_eq_electric_magnetic {d} {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (hA : ContDiff ℝ 2 A)
    (J : LorentzCurrentDensity d) (x : SpaceTime d) :
    A.lagrangian 𝓕 J x = 1 / 2 * (𝓕.ε₀ * ‖A.electricField 𝓕.c (x.time 𝓕.c) x.space‖ ^ 2 -
    (1 / (2 * 𝓕.μ₀)) * ∑ i, ∑ j, ‖A.magneticFieldMatrix 𝓕.c (x.time 𝓕.c) x.space (i, j)‖ ^ 2)
    - A.scalarPotential 𝓕.c (x.time 𝓕.c) x.space * J.chargeDensity 𝓕.c (x.time 𝓕.c) x.space
    + ∑ i, A.vectorPotential 𝓕.c (x.time 𝓕.c) x.space i *
        J.currentDensity 𝓕.c (x.time 𝓕.c) x.space i
```

lemma lagrangian_hasVarGradientAt_eq_add_gradKineticTerm [source]

```
lemma lagrangian_hasVarGradientAt_eq_add_gradKineticTerm {𝓕 : FreeSpace}
    (A : ElectromagneticPotential d) (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) :
    HasVarGradientAt (fun A => lagrangian 𝓕 A J)
    (A.gradKineticTerm 𝓕 - A.gradFreeCurrentPotential J) A
```

def gradLagrangian [source]

The variational gradient of the lagrangian of electromagnetic field.

```
noncomputable def gradLagrangian {d} (𝓕 : FreeSpace) (A : ElectromagneticPotential d)
    (J : LorentzCurrentDensity d) : SpaceTime d → Lorentz.Vector d
```

lemma gradLagrangian_eq_kineticTerm_sub [source]

```
lemma gradLagrangian_eq_kineticTerm_sub {𝓕 : FreeSpace} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) :
    A.gradLagrangian 𝓕 J = A.gradKineticTerm 𝓕 - A.gradFreeCurrentPotential J
```

lemma lagrangian_hasVarGradientAt_gradLagrangian [source]

```
lemma lagrangian_hasVarGradientAt_gradLagrangian {𝓕 : FreeSpace} (A : ElectromagneticPotential
     d)
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d) (hJ : ContDiff ℝ ∞ J) :
    HasVarGradientAt (fun A => lagrangian 𝓕 A J) (A.gradLagrangian 𝓕 J) A
```

lemma gradLagrangian_eq_sum_fieldStrengthMatrix [source]

```
lemma gradLagrangian_eq_sum_fieldStrengthMatrix {𝓕 : FreeSpace} (A : ElectromagneticPotential
    d)
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d) (hJ : ContDiff ℝ ∞ J) :
    A.gradLagrangian 𝓕 J = fun x => ∑ ν,
      (η ν ν • (1 / 𝓕.μ₀ * ∑ μ, ∂_ μ (fun x => (A.fieldStrengthMatrix x) (μ, ν)) x - J x ν)
      • Lorentz.Vector.basis ν)
```

lemma gradLagrangian_eq_electricField_magneticField [source]

```
lemma gradLagrangian_eq_electricField_magneticField {𝓕 : FreeSpace} (A :
    ElectromagneticPotential d)
```

```
    (hA : ContDiff ℝ ∞ A) (J : LorentzCurrentDensity d)
    (hJ : ContDiff ℝ ∞ J) (x : SpaceTime d) :
    A.gradLagrangian 𝓕 J x = (1 / (𝓕.μ₀ * 𝓕.c.val) *
        Space.div (electricField 𝓕.c A ((time 𝓕.c) x)) (space x) +
        - 𝓕.c * J.chargeDensity 𝓕.c (x.time 𝓕.c) x.space) •
      Lorentz.Vector.basis (Sum.inl 0) +
    ∑ i, (𝓕.μ₀⁻¹ * (𝓕.ε₀ * 𝓕.μ₀ * ∂ₜ (electricField 𝓕.c A · x.space) ((time 𝓕.c) x) i -
      ∑ j, ∂[j] (magneticFieldMatrix 𝓕.c A (x.time 𝓕.c) · (j, i)) x.space) +
      J.currentDensity 𝓕.c (x.time 𝓕.c) x.space i) •
        Lorentz.Vector.basis (Sum.inr i)
```

2.13 PhysLean.Electromagnetism.Kinematics.Boosts

PhysLean/Electromagnetism/Kinematics/Boosts.lean on GitHub.

lemma electricField_apply_x_boost_zero [source]

```
lemma electricField_apply_x_boost_zero {d : ℕ} {c : SpeedOfLight} (β : ℝ) (hβ : |β| < 1)
    (A : ElectromagneticPotential d.succ) (hA : Differentiable ℝ A) (t : Time) (x : Space d.
    succ) :
    let Λ
```

lemma electricField_apply_x_boost_succ [source]

```
lemma electricField_apply_x_boost_succ {d : ℕ} {c : SpeedOfLight} (β : ℝ) (hβ : |β| < 1)
    (A : ElectromagneticPotential d.succ) (hA : Differentiable ℝ A) (t : Time) (x : Space d.
    succ)
    (i : Fin d) :
    let Λ
```

lemma magneticFieldMatrix_apply_x_boost_zero_succ [source]

```
lemma magneticFieldMatrix_apply_x_boost_zero_succ {d : ℕ} {c : SpeedOfLight} (β : ℝ) (hβ : |β|
    < 1)
    (A : ElectromagneticPotential d.succ) (hA : Differentiable ℝ A) (t : Time) (x : Space d.
    succ)
    (i : Fin d) :
    let Λ
```

lemma magneticFieldMatrix_apply_x_boost_succ_succ [source]

```
lemma magneticFieldMatrix_apply_x_boost_succ_succ {d : ℕ} {c : SpeedOfLight} (β : ℝ) (hβ : |β|
    < 1)
    (A : ElectromagneticPotential d.succ) (hA : Differentiable ℝ A) (t : Time) (x : Space d.
    succ)
    (i j : Fin d) :
    let Λ
```

2.14 PhysLean.Electromagnetism.Kinematics.ElectricField

PhysLean/Electromagnetism/Kinematics/ElectricField.lean on GitHub.

lemma electricField_eq [source]

```
lemma electricField_eq {c : SpeedOfLight} (A : ElectromagneticPotential d) :
    A.electricField c = fun t x =>
      - ∇ (A.scalarPotential c t) x - ∂ₜ (fun t => A.vectorPotential c t x) t
```

lemma fieldStrengthMatrix_inl_inr_eq_electricField [source]

```
lemma fieldStrengthMatrix_inl_inr_eq_electricField {c : SpeedOfLight}
    (A : ElectromagneticPotential d)
    (x : SpaceTime d) (i : Fin d) (hA : Differentiable ℝ A) :
    A.fieldStrengthMatrix x (Sum.inl 0, Sum.inr i) =
    - (1 /c) * A.electricField c (x.time c) x.space i
```

lemma fieldStrengthMatrix_inr_inl_eq_electricField [source]

```
lemma fieldStrengthMatrix_inr_inl_eq_electricField {c : SpeedOfLight}
    (A : ElectromagneticPotential d)
    (x : SpaceTime d) (i : Fin d) (hA : Differentiable ℝ A) :
    A.fieldStrengthMatrix x (Sum.inr i, Sum.inl 0) =
    (1 /c) * A.electricField c (x.time c) x.space i
```

lemma electricField_contDiff [source]

```
lemma electricField_contDiff {n} {c : SpeedOfLight} {A : ElectromagneticPotential d}
    (hA : ContDiff ℝ (n + 1) A) : ContDiff ℝ n ↿(A.electricField c)
```

lemma electricField_apply_contDiff [source]

```
lemma electricField_apply_contDiff {n} {c : SpeedOfLight} {A : ElectromagneticPotential d}
    (hA : ContDiff ℝ (n + 1) A) : ContDiff ℝ n (↿(fun t x => A.electricField c t x i))
```

lemma electricField_apply_contDiff_space [source]

```
lemma electricField_apply_contDiff_space {n} {A : ElectromagneticPotential d}
    {c : SpeedOfLight}
    (hA : ContDiff ℝ (n + 1) A) (t : Time) :
    ContDiff ℝ n (fun x => A.electricField c t x i)
```

lemma electricField_apply_contDiff_time [source]

```
lemma electricField_apply_contDiff_time {n} {c : SpeedOfLight} {A : ElectromagneticPotential d}
    (hA : ContDiff ℝ (n + 1) A) (x : Space d) :
    ContDiff ℝ n (fun t => A.electricField c t x i)
```

lemma electricField_differentiable_time [source]

```
lemma electricField_differentiable_time {A : ElectromagneticPotential d} {c : SpeedOfLight}
    (hA : ContDiff ℝ 2 A) (x : Space d) : Differentiable ℝ (A.electricField c · x)
```

lemma electricField_differentiable_space [source]

```
lemma electricField_differentiable_space {A : ElectromagneticPotential d} {c : SpeedOfLight}
    (hA : ContDiff ℝ 2 A) (t : Time) : Differentiable ℝ (A.electricField c t)
```

lemma electricField_apply_differentiable_space [source]

```
lemma electricField_apply_differentiable_space {A : ElectromagneticPotential d}
    {c : SpeedOfLight}
    (hA : ContDiff ℝ 2 A) (t : Time) (i : Fin d) :
    Differentiable ℝ (fun x => A.electricField c t x i)
```

lemma time_deriv_vectorPotential_eq_electricField [source]

```
lemma time_deriv_vectorPotential_eq_electricField {d} {c : SpeedOfLight}
    (A : ElectromagneticPotential d)
    (t : Time) (x : Space d) :
    ∂ₜ (fun t => A.vectorPotential c t x) t =
    - A.electricField c t x - ∇ (A.scalarPotential c t) x
```

lemma time_deriv_comp_vectorPotential_eq_electricField [source]

```
lemma time_deriv_comp_vectorPotential_eq_electricField {d} {A : ElectromagneticPotential d}
    {c : SpeedOfLight}
    (hA : Differentiable ℝ A)
    (t : Time) (x : Space d) (i : Fin d) :
    ∂ₜ (fun t => A.vectorPotential c t x i) t =
    - A.electricField c t x i - ∂[i] (A.scalarPotential c t) x
```

lemma time_deriv_electricField_eq_fieldStrengthMatrix [source]

```
lemma time_deriv_electricField_eq_fieldStrengthMatrix {d} {A : ElectromagneticPotential d}
    {c : SpeedOfLight} (hA : ContDiff ℝ 2 A) (t : Time) (x : Space d) (i : Fin d) :
    ∂ₜ (fun t => A.electricField c t x) t i =
    - c ^ 2 * ∂_ (Sum.inl 0) (fun x => (A.fieldStrengthMatrix x) (Sum.inl 0, Sum.inr i))
    ((toTimeAndSpace c).symm (t, x))
```

lemma div_electricField_eq_fieldStrengthMatrix [source]

```
lemma div_electricField_eq_fieldStrengthMatrix{d} {A : ElectromagneticPotential d}
    {c : SpeedOfLight} (hA : ContDiff ℝ 2 A) (t : Time) (x : Space d) :
    (∇ · A.electricField c t) x = c * ∑ (μ : (Fin 1 ⊕ Fin d)),
      (∂_ μ (A.fieldStrengthMatrix · (μ, Sum.inl 0)) ((toTimeAndSpace c).symm (t, x)))
```

2.15 PhysLean.Electromagnetism.Kinematics.EMPotential

PhysLean/Electromagnetism/Kinematics/EMPotential.lean on GitHub.

instance InnerProductSpace ℝ (SpaceTime d) [source]

A local instance of the inner product structure on `SpaceTime`.

```
noncomputable local instance {d : ℕ}: InnerProductSpace ℝ (SpaceTime d)
```

2.16 PhysLean.Electromagnetism.Kinematics.FieldStrength

PhysLean/Electromagnetism/Kinematics/FieldStrength.lean on GitHub.

lemma fieldStrengthMatrix_differentiable_space [source]

```
lemma fieldStrengthMatrix_differentiable_space {d} {A : ElectromagneticPotential d}
    {μν} (hA : ContDiff ℝ 2 A) (t : Time) {c : SpeedOfLight} :
    Differentiable ℝ (fun x => A.fieldStrengthMatrix ((toTimeAndSpace c).symm (t, x)) μν)
```

lemma fieldStrengthMatrix_differentiable_time [source]

```
lemma fieldStrengthMatrix_differentiable_time {d} {A : ElectromagneticPotential d}
    {μν} (hA : ContDiff ℝ 2 A) (x : Space d) {c : SpeedOfLight} :
    Differentiable ℝ (fun t => A.fieldStrengthMatrix ((toTimeAndSpace c).symm (t, x)) μν)
```

lemma fieldStrengthMatrix_contDiff [source]

```
lemma fieldStrengthMatrix_contDiff {d} {n : WithTop ℕ∞} {A : ElectromagneticPotential d}
    {μν} (hA : ContDiff ℝ (n + 1) A) :
    ContDiff ℝ n (A.fieldStrengthMatrix · μν)
```

lemma fieldStrengthMatrix_smooth [source]

```
lemma fieldStrengthMatrix_smooth {d} {A : ElectromagneticPotential d}
    (hA : ContDiff ℝ ∞ A) (μν) :
    ContDiff ℝ ∞ (A.fieldStrengthMatrix · μν)
```

lemma toFieldStrength_add [source]

```
lemma toFieldStrength_add {d} (A1 A2 : ElectromagneticPotential d)
    (x : SpaceTime d) (hA1 : Differentiable ℝ A1) (hA2 : Differentiable ℝ A2) :
    toFieldStrength (A1 + A2) x = toFieldStrength A1 x + toFieldStrength A2 x
```

lemma fieldStrengthMatrix_add [source]

```
lemma fieldStrengthMatrix_add {d} (A1 A2 : ElectromagneticPotential d)
    (x : SpaceTime d) (hA1 : Differentiable ℝ A1) (hA2 : Differentiable ℝ A2) :
    (A1 + A2).fieldStrengthMatrix x =
    A1.fieldStrengthMatrix x + A2.fieldStrengthMatrix x
```

lemma toFieldStrength_smul [source]

```
lemma toFieldStrength_smul {d} (c : ℝ) (A : ElectromagneticPotential d)
    (x : SpaceTime d) (hA : Differentiable ℝ A) :
    toFieldStrength (c • A) x = c • toFieldStrength A x
```

lemma fieldStrengthMatrix_smul [source]

```
lemma fieldStrengthMatrix_smul {d} (c : ℝ) (A : ElectromagneticPotential d)
    (x : SpaceTime d) (hA : Differentiable ℝ A) :
    (c • A).fieldStrengthMatrix x = c • A.fieldStrengthMatrix x
```

2.17 PhysLean.Electromagnetism.Kinematics.MagneticField

PhysLean/Electromagnetism/Kinematics/MagneticField.lean on GitHub.

lemma magneticField_eq [source]

```
lemma magneticField_eq {c : SpeedOfLight} (A : ElectromagneticPotential) :
    A.magneticField c = fun t x => (∇ × (A.vectorPotential c t)) x
```

lemma magneticField_div_eq_zero [source]

```
lemma magneticField_div_eq_zero (A : ElectromagneticPotential)
    (hA : ContDiff ℝ 2 A) (t : Time) : Space.div (A.magneticField c t) = 0
```

def magneticFieldMatrix [source]

The matrix corresponding to the magnetic field in general dimensions. In 3 space-dimensions this reduces to a vector.

```
noncomputable def magneticFieldMatrix (c : SpeedOfLight := 1) (A : ElectromagneticPotential d)
    :
    Time → Space d → (Fin d × Fin d) → ℝ
```

lemma magneticFieldMatrix_eq [source]

```
lemma magneticFieldMatrix_eq {c : SpeedOfLight} (A : ElectromagneticPotential d) :
    A.magneticFieldMatrix c = fun t x ij =>
      A.fieldStrengthMatrix ((toTimeAndSpace c).symm (t, x)) (Sum.inr ij.1, Sum.inr ij.2)
```

lemma fieldStrengthMatrix_inr_inr_eq_magneticFieldMatrix [source]

```
lemma fieldStrengthMatrix_inr_inr_eq_magneticFieldMatrix {c : SpeedOfLight}
    (A : ElectromagneticPotential d)
    (x : SpaceTime d) (i j : Fin d) :
    A.fieldStrengthMatrix x (Sum.inr i, Sum.inr j) =
    A.magneticFieldMatrix c (x.time c) x.space (i, j)
```

lemma magneticFieldMatrix_antisymm [source]

```
lemma magneticFieldMatrix_antisymm {c : SpeedOfLight}
    (A : ElectromagneticPotential d) (t : Time)
    (x : Space d) (i j : Fin d) :
    A.magneticFieldMatrix c t x (i, j) = - A.magneticFieldMatrix c t x (j, i)
```

lemma magneticFieldMatrix_diag_eq_zero [source]

```
@[simp]
lemma magneticFieldMatrix_diag_eq_zero {c : SpeedOfLight}
    (A : ElectromagneticPotential d) (t : Time)
    (x : Space d) (i : Fin d) :
    A.magneticFieldMatrix c t x (i, i) = 0
```

lemma magneticField_eq_magneticFieldMatrix [source]

```
lemma magneticField_eq_magneticFieldMatrix {c : SpeedOfLight} (A : ElectromagneticPotential)
    (hA : Differentiable ℝ A) :
    A.magneticField c = fun t x i =>
      match i with
      | 0 => - A.magneticFieldMatrix c t x (1, 2)
      | 1 => A.magneticFieldMatrix c t x (0, 2)
      | 2 => - A.magneticFieldMatrix c t x (0, 1)
```

lemma magneticField_curl_eq_magneticFieldMatrix [source]

```
lemma magneticField_curl_eq_magneticFieldMatrix{c : SpeedOfLight} (A : ElectromagneticPotential
    )
    (hA : ContDiff ℝ 2 A) (t : Time) :
    (∇ × A.magneticField c t) x i = ∑ j, Space.deriv j (A.magneticFieldMatrix c t · (j, i)) x
```

lemma magneticFieldMatrix_eq_vectorPotential [source]

```
lemma magneticFieldMatrix_eq_vectorPotential {c : SpeedOfLight} (A : ElectromagneticPotential d
    )
    (hA : Differentiable ℝ A) (t : Time) (x : Space d) (i j : Fin d) :
    A.magneticFieldMatrix c t x (i, j) = Space.deriv j (A.vectorPotential c t · i) x -
    Space.deriv i (A.vectorPotential c t · j) x
```

lemma magneticFieldMatrix_contDiff [source]

```
lemma magneticFieldMatrix_contDiff {n} {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ (n + 1) A) (ij) :
    ContDiff ℝ n ↿(fun t x => A.magneticFieldMatrix c t x ij)
```

lemma magneticFieldMatrix_space_contDiff [source]

```
lemma magneticFieldMatrix_space_contDiff {n} {c : SpeedOfLight} (A : ElectromagneticPotential d
    )
    (hA : ContDiff ℝ (n + 1) A) (t : Time) (ij) :
    ContDiff ℝ n (fun x => A.magneticFieldMatrix c t x ij)
```

lemma magneticFieldMatrix_time_contDiff [source]

```
lemma magneticFieldMatrix_time_contDiff {n} {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ (n + 1) A) (x : Space d) (ij) :
    ContDiff ℝ n (fun t => A.magneticFieldMatrix c t x ij)
```

lemma magneticFieldMatrix_differentiable [source]

```
lemma magneticFieldMatrix_differentiable {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 2 A) (ij) : Differentiable ℝ ↿(fun t x => A.magneticFieldMatrix c t x ij)
```

lemma magneticFieldMatrix_differentiable_space [source]

```
lemma magneticFieldMatrix_differentiable_space {c : SpeedOfLight} (A : ElectromagneticPotential
     d)
```

```
    (hA : ContDiff ℝ 2 A) (t : Time) (ij) :
    Differentiable ℝ (fun x => A.magneticFieldMatrix c t x ij)
```

lemma magneticFieldMatrix_differentiable_time [source]

```
lemma magneticFieldMatrix_differentiable_time {c : SpeedOfLight} (A : ElectromagneticPotential
    d)
    (hA : ContDiff ℝ 2 A) (x : Space d) (ij) :
    Differentiable ℝ (fun t => A.magneticFieldMatrix c t x ij)
```

lemma magneticFieldMatrix_space_deriv_eq [source]

```
lemma magneticFieldMatrix_space_deriv_eq {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 2 A) (t : Time) (x : Space d) (i j k : Fin d) :
    ∂[k] (A.magneticFieldMatrix c t · (i, j)) x =
    ∂[i] (A.magneticFieldMatrix c t · (k, j)) x
    - ∂[j] (A.magneticFieldMatrix c t · (k, i)) x
```

lemma time_deriv_magneticFieldMatrix [source]

```
lemma time_deriv_magneticFieldMatrix {d : ℕ} {c : SpeedOfLight} (A : ElectromagneticPotential d
    )
    (hA : ContDiff ℝ 2 A) (t : Time) (x : Space d) (i j : Fin d) :
    ∂ₜ (A.magneticFieldMatrix c · x (i, j)) t =
    ∂[i] (A.electricField c t · j) x - ∂[j] (A.electricField c t · i) x
```

lemma time_deriv_time_deriv_magneticFieldMatrix [source]

```
lemma time_deriv_time_deriv_magneticFieldMatrix {d : ℕ} {c : SpeedOfLight}
    (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 3 A) (t : Time) (x : Space d) (i j : Fin d) :
    ∂ₜ (∂ₜ (A.magneticFieldMatrix c · x (i, j))) t =
    ∂[i] (fun x => ∂ₜ (fun t => A.electricField c t x) t j) x -
    ∂[j] (fun x => ∂ₜ (fun t => A.electricField c t x) t i) x
```

lemma curl_magneticFieldMatrix_eq_electricField_fieldStrengthMatrix [source]

```
lemma curl_magneticFieldMatrix_eq_electricField_fieldStrengthMatrix {d : ℕ} {c : SpeedOfLight}
    (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ 2 A) (t : Time) (x : Space d) (i : Fin d) :
    ∑ j, Space.deriv j (A.magneticFieldMatrix c t · (j, i)) x =
    (1/c^2) * ∂ₜ (fun t => A.electricField c t x) t i +
    (∑ (μ : (Fin 1 ⊕ Fin d)), (∂_ μ (A.fieldStrengthMatrix · (μ, Sum.inr i))
    ((toTimeAndSpace c).symm (t, x))))
```

2.18 PhysLean.Electromagnetism.Kinematics.ScalarPotential

PhysLean/Electromagnetism/Kinematics/ScalarPotential.lean on GitHub.

lemma scalarPotential_contDiff [source]

```
lemma scalarPotential_contDiff {n} {d} (c : SpeedOfLight) (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ n A) : ContDiff ℝ n ⇑(A.scalarPotential c)
```

lemma scalarPotential_contDiff_space [source]

```
lemma scalarPotential_contDiff_space {n} {d} (c : SpeedOfLight)
    (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ n A) (t : Time) : ContDiff ℝ n (A.scalarPotential c t)
```

lemma scalarPotential_contDiff_time [source]

```
lemma scalarPotential_contDiff_time {n} {d} (c : SpeedOfLight) (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ n A) (x : Space d) : ContDiff ℝ n (A.scalarPotential c · x)
```

lemma scalarPotential_differentiable [source]

```
lemma scalarPotential_differentiable {d} (c : SpeedOfLight) (A : ElectromagneticPotential d)
    (hA : Differentiable ℝ A) : Differentiable ℝ ↿(A.scalarPotential c)
```

lemma scalarPotential_differentiable_space [source]

```
lemma scalarPotential_differentiable_space {d} (c : SpeedOfLight) (A : ElectromagneticPotential
     d)
    (hA : Differentiable ℝ A) (t : Time) : Differentiable ℝ (A.scalarPotential c t)
```

lemma scalarPotential_differentiable_time [source]

```
lemma scalarPotential_differentiable_time {d} (c : SpeedOfLight) (A : ElectromagneticPotential
    d)
    (hA : Differentiable ℝ A) (x : Space d) : Differentiable ℝ (A.scalarPotential c · x)
```

2.19 PhysLean.Electromagnetism.Kinematics.VectorPotential

PhysLean/Electromagnetism/Kinematics/VectorPotential.lean on GitHub.

lemma vectorPotential_contDiff [source]

```
lemma vectorPotential_contDiff {n} {d} {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ n A) : ContDiff ℝ n ↿(A.vectorPotential c)
```

lemma vectorPotential_apply_contDiff [source]

```
lemma vectorPotential_apply_contDiff {n} {d} {c : SpeedOfLight} (A : ElectromagneticPotential d
    )
    (hA : ContDiff ℝ n A) (i : Fin d) : ContDiff ℝ n ↿(fun t x => A.vectorPotential c t x i)
```

lemma vectorPotential_comp_contDiff [source]

```
lemma vectorPotential_comp_contDiff {n} {d} {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ n A) (i : Fin d) : ContDiff ℝ n ↿(fun t x => A.vectorPotential c t x i)
```

lemma vectorPotential_contDiff_space [source]

```
lemma vectorPotential_contDiff_space {n} {d} {c : SpeedOfLight} (A : ElectromagneticPotential d
    )
```

```
    (hA : ContDiff ℝ n A) (t : Time) : ContDiff ℝ n (A.vectorPotential c t)
```

lemma vectorPotential_apply_contDiff_space [source]

```
lemma vectorPotential_apply_contDiff_space {n} {d} {c : SpeedOfLight}
    (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ n A) (t : Time) (i : Fin d) :
    ContDiff ℝ n (fun x => A.vectorPotential c t x i)
```

lemma vectorPotential_contDiff_time [source]

```
lemma vectorPotential_contDiff_time {n} {d} {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : ContDiff ℝ n A) (x : Space d) : ContDiff ℝ n (A.vectorPotential c · x)
```

lemma vectorPotential_differentiable [source]

```
lemma vectorPotential_differentiable {d} {c : SpeedOfLight} (A : ElectromagneticPotential d)
    (hA : Differentiable ℝ A) : Differentiable ℝ ↿(A.vectorPotential c)
```

lemma vectorPotential_differentiable_time [source]

```
lemma vectorPotential_differentiable_time {d} {c : SpeedOfLight} (A : ElectromagneticPotential
    d)
    (hA : Differentiable ℝ A) (x : Space d) : Differentiable ℝ (A.vectorPotential c · x)
```

2.20 PhysLean.Electromagnetism.PointParticle.OneDimension

PhysLean/Electromagnetism/PointParticle/OneDimension.lean on GitHub.

def oneDimPointParticleCurrentDensity [source]

The current density of of a point particle stationary at the origin of 1d space.

```
noncomputable def oneDimPointParticleCurrentDensity (q : ℝ) : LorentzCurrentDensityD 1
```

def oneDimPointParticle [source]

The electromagnetic potential of a point particle stationary at the origin of 1d space.

```
noncomputable def oneDimPointParticle (q : ℝ) : ElectromagneticPotentialD 1
```

lemma oneDimPointParticle_vectorPotential [source]

```
@[simp]
lemma oneDimPointParticle_vectorPotential (q : ℝ) :
    (oneDimPointParticle q).vectorPotential = 0
```

lemma oneDimPointParticle_scalarPotential [source]

```
lemma oneDimPointParticle_scalarPotential (q : ℝ) :
    (oneDimPointParticle q).scalarPotential =
    Space.constantTime (- Distribution.ofFunction (fun x => (q/(2)) • ‖x‖)
```

```
    (by
      apply IsDistBounded.const_smul
      convert IsDistBounded.pow (n := 1) (by simp)
      simp)
    (by fun_prop))
```

lemma oneDimPointParticle_electricField_eq_heavisideStep [source]

```
lemma oneDimPointParticle_electricField_eq_heavisideStep (q : ℝ) :
    (oneDimPointParticle q).electricField = constantTime (q •
    ((heavisideStep 0).smulRight (basis 0) - (1 / (2 : ℝ)) • constD 1 (basis 0)))
```

lemma oneDimPointParticle_gaussLaw [source]

```
lemma oneDimPointParticle_gaussLaw (q : ℝ) :
    spaceDivD (oneDimPointParticle q).electricField = constantTime (q • diracDelta ℝ 0)
```

lemma oneDimPointParticle_gradLagrangian [source]

```
lemma oneDimPointParticle_gradLagrangian (q : ℝ) :
    (oneDimPointParticle q).gradLagrangian (oneDimPointParticleCurrentDensity q) = 0
```

2.21 PhysLean.Electromagnetism.Vacuum.Constant

PhysLean/Electromagnetism/Vacuum/Constant.lean on GitHub.

def constantEB [source]

An electric potential which gives a given constant E-field and B-field.

```
@[nolint unusedArguments]
noncomputable def constantEB {d : ℕ} (c : SpeedOfLight)
    (E₀ : EuclideanSpace ℝ (Fin d)) (B₀ : Fin d × Fin d → ℝ)
    (B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)) : ElectromagneticPotential d
```

lemma constantEB_smooth [source]

```
lemma constantEB_smooth {c : SpeedOfLight}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} :
    ContDiff ℝ ∞ (constantEB c E₀ B₀ B₀_antisymm)
```

lemma constantEB_scalarPotential [source]

```
lemma constantEB_scalarPotential {c : SpeedOfLight}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} :
    (constantEB c E₀ B₀ B₀_antisymm).scalarPotential c = fun _ x => -⟪E₀, x⟫_ℝ
```

lemma constantEB_vectorPotential [source]

```
lemma constantEB_vectorPotential {c : SpeedOfLight}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
```

```
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} :
    (constantEB c E₀ B₀ B₀_antisymm).vectorPotential c = fun _ x i =>
      (1 / 2) * ∑ j, B₀ (i, j) * x j
```

lemma constantEB_vectorPotential_time_deriv [source]

```
@[simp]
lemma constantEB_vectorPotential_time_deriv {c : SpeedOfLight}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} (t : Time) (x : Space d) :
    ∂ₜ ((constantEB c E₀ B₀ B₀_antisymm).vectorPotential c · x) t = 0
```

lemma constantEB_vectorPotential_space_deriv [source]

```
lemma constantEB_vectorPotential_space_deriv {c : SpeedOfLight}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} (t : Time) (x : Space d) (i j : Fin d) :
    Space.deriv i ((constantEB c E₀ B₀ B₀_antisymm).vectorPotential c t · j) x =
    (1 / 2) * B₀ (j, i)
```

lemma constantEB_electricField [source]

```
@[simp]
lemma constantEB_electricField {c : SpeedOfLight}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} :
    (constantEB c E₀ B₀ B₀_antisymm).electricField c = fun _ _ => E₀
```

lemma constantEB_magneticFieldMatrix [source]

```
@[simp]
lemma constantEB_magneticFieldMatrix {c : SpeedOfLight}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} :
    (constantEB c E₀ B₀ B₀_antisymm).magneticFieldMatrix c = fun _ _ => B₀
```

lemma constantEB_isExtrema [source]

```
lemma constantEB_isExtrema {ℱ : FreeSpace}
    {E₀ : EuclideanSpace ℝ (Fin d)} {B₀ : Fin d × Fin d → ℝ}
    {B₀_antisymm : ∀ i j, B₀ (i, j) = - B₀ (j, i)} :
    IsExtrema ℱ (constantEB ℱ.c E₀ B₀ B₀_antisymm) 0
```

2.22 PhysLean.Mathematics.RatComplexNum

PhysLean/Mathematics/RatComplexNum.lean on GitHub.

instance Add RatComplexNum [source]

```
instance : Add RatComplexNum where
```

instance Mul RatComplexNum [source]

```
instance : Mul RatComplexNum where
```

2.23 PhysLean.Mathematics.VariationalCalculus.HasVarAdjDeriv

PhysLean/Mathematics/VariationalCalculus/HasVarAdjDeriv.lean on GitHub.

lemma fun_mul [source]

```
lemma fun_mul {f : X → ℝ} (hf : ContDiff ℝ ∞ f)
    (F : (X → U) → (X → ℝ)) (F') (u)
    (hF : HasVarAdjDerivAt F F' u) :
    HasVarAdjDerivAt (fun φ x => f x * F φ x) (fun ψ x => F' (fun x' => f x' * ψ x') x) u
```

2.24 PhysLean.Mathematics.VariationalCalculus.HasVarGradient

PhysLean/Mathematics/VariationalCalculus/HasVarGradient.lean on GitHub.

lemma add [source]

```
lemma HasVarGradientAt.add (F F' : (X → U) → (X → ℝ))
    {grad grad' : X → U} {u : X → U} [OpensMeasurableSpace X]
    [IsFiniteMeasureOnCompacts (@volume X _)]
    (h : HasVarGradientAt F grad u) (h' : HasVarGradientAt F' grad' u) :
    HasVarGradientAt (F + F') (grad + grad') u
```

lemma sum [source]

```
lemma HasVarGradientAt.sum {ι : Type} [Fintype ι] (F : ι → (X → U) → (X → ℝ))
    {grad : ι → X → U} {u : X → U} (hu : ContDiff ℝ ∞ u) [OpensMeasurableSpace X]
    [IsFiniteMeasureOnCompacts (@volume X _)]
    (h : ∀ i, HasVarGradientAt (F i) (grad i) u) :
    HasVarGradientAt (fun v x => ∑ i, F i v x) (∑ i, grad i) u
```

lemma neg [source]

```
lemma HasVarGradientAt.neg {F : (X → U) → (X → ℝ)}
    {grad : X → U} {u : X → U}
    (h : HasVarGradientAt F grad u) :
    HasVarGradientAt (-F) (-grad) u
```

2.25 PhysLean.Meta.AllFilePaths

PhysLean/Meta/AllFilePaths.lean on GitHub.

def allPhysLeanModules [source]

Gets an array of all module names in `PhysLean`.

```
def allPhysLeanModules : IO (Array Lean.Name)
```

2.26 PhysLean.Meta.Basic

PhysLean/Meta/Basic.lean on GitHub.

def allUserConsts [source]

All user defined declarations.

```
def allUserConsts : CoreM (Array ConstantInfo)
```

2.27 PhysLean.Meta.Sorry

PhysLean/Meta/Sorry.lean on GitHub.

structure State [source]

A structure used for collecting names of results dependent on the `sorryAx` axiom and the `Lean.ofReduceBool` axiom.

```
structure State where
  visited : NameSet := {}
  containsSorry : NameSet := {}
  containsOfReduceBool : NameSet := {}
```

abbrev M [source]

A monad used for collecting names of results dependent on the `sorryAx` axiom and the `Lean.ofReduceBool` axiom.

```
abbrev M
```

def collect [source]

Given a `c : Name` updating the monad `M` based on which results which `c` uses depend on the `sorryAx` axiom and the `Lean.ofReduceBool` axiom.

```
partial def collect (c : Name) (parents : NameSet) : M Unit
```

def allSorryPseudo [source]

Given a `c : Array Name` updating the monad `M` based on which results depend on the `sorryAx` axiom and the `Lean.ofReduceBool` axiom.

```
partial def allSorryPseudo (c : Array Name) : M Unit
```

def collectSorryPseudo [source]

Given a `c : Array Name` the names of all results used to defined the results in `c` with the `sorryAx` axiom and the `Lean.ofReduceBool` axiom.

```
def collectSorryPseudo (c : Array Name) : CoreM (Array Name × Array Name)
```

def allWithSorryPseudo [source]

The axioms of a constant.

```
def allWithSorryPseudo : CoreM (Array Name × Array Name)
```

def allSorryfulAttributed [source]

All names which are attributed `sorryful`.

```
unsafe def allSorryfulAttributed : CoreM (Array Name)
```

def allPseudoAttributed [source]

All names which are attributed `pseudo`.

```
unsafe def allPseudoAttributed : CoreM (Array Name)
```

def sorryfulPseudoTest [source]

Checks whether all results attributed `sorryful` depend on the ``sorryAx` axiom and vice versa.

```
unsafe def sorryfulPseudoTest : MetaM Unit
```

2.28 PhysLean.Particles.NeutrinoPhysics.Basic

PhysLean/Particles/NeutrinoPhysics/Basic.lean on GitHub.

def diagPhase [source]

diagonal phase matrix from a real-valued function on indices

```
@[simp]
def diagPhase (θ : Fin 3 → ℝ) : Matrix (Fin 3) (Fin 3) ℂ
```

lemma diagPhase_zero [source]

lemma stating that the diagonal phase matrix with all zeros is the identity matrix

```
@[simp]
lemma diagPhase_zero:
    diagPhase (fun _ : Fin 3 => 0) = 1
```

lemma diagPhase_zero_eq [source]

lemma stating that diagPhase with $\theta = 0$ is equal to diagPhase with all zero phases

```
@[simp]
lemma diagPhase_zero_eq : diagPhase 0 = diagPhase (fun _ : Fin 3 => 0)
```

lemma diagPhase_star [source]

lemma stating that the Hermitian conjugate of diagPhase diag(+iθ_i) is just diagPhase with entries diag(-iθ_i)

```
@[simp]
lemma diagPhase_star (θ : Fin 3 → ℝ) :
    (diagPhase θ)^H = diagPhase (- θ)
```

lemma diagPhase_mul [source]

lemma stating that multiplying two phase matrices is equivalent to adding the phases

```
@[simp]
lemma diagPhase_mul (θ φ : Fin 3 → ℝ) :
    diagPhase θ * diagPhase φ = diagPhase (θ + φ)
```

def diagPhase_unitary [source]

diagonal phase matrix diag(iθ_i) is part of the unitary group

```
def diagPhase_unitary (θ : Fin 3 → ℝ) : unitaryGroup (Fin 3) ℂ
```

lemma diagPhaseShift_coe_matrix [source]

The underlying matrix of the phase-shift element of the unitary group is the phase-shift matrix.

```
@[simp]
lemma diagPhaseShift_coe_matrix (θ : Fin 3 → ℝ) : ↑(diagPhase_unitary θ) = diagPhase θ
```

def leptonPhaseShift [source]

The Lepton phase shift matrix as a `3×3` complex matrix, given three reals `a b c`. This dictates the phase shift freedom of the charged lepton sector.

```
def leptonPhaseShift (a b c : ℝ) : Matrix (Fin 3) (Fin 3) ℂ
```

def neutrinoPhaseShift [source]

The neutrino phase shift matrix as a `3×3` complex matrix, given three reals `d e f`. This dictates the phase shift freedom of the neutrino sector (If neutrinos are Dirac).

```
def neutrinoPhaseShift (d e f : ℝ) : Matrix (Fin 3) (Fin 3) ℂ
```

def majoranaPhaseMatrix [source]

If neutrinos are Majorana particles, then the neutrino phase shift matrix is physical, and cannot be absorbed into the definition of the neutrino fields.

```
def majoranaPhaseMatrix (α1 α2 : ℝ) : Matrix (Fin 3) (Fin 3) ℂ
```

def PMNS_dirac_equivalence [source]

The Dirac PMNS matrix equivalence relations

```
def PMNS_dirac_equivalence (U V : unitaryGroup (Fin 3) ℂ) : Prop
```

lemma PMNS_dirac_equivalence_refl [source]

The relation `PMNS_dirac_equivalence` is reflexive.

```
lemma PMNS_dirac_equivalence_refl :
    ∀ U : unitaryGroup (Fin 3) ℂ, PMNS_dirac_equivalence U U
```

lemma PMNS_dirac_equivalence_symm [source]

The relation `PMNS_dirac_equivalence` is symmetric.

```
lemma PMNS_dirac_equivalence_symm :
    ∀ U V : unitaryGroup (Fin 3) ℂ, PMNS_dirac_equivalence U V → PMNS_dirac_equivalence V U
```

lemma PMNS_dirac_equivalence_trans [source]

The relation `PMNS_dirac_equivalence` is transitive.

```
lemma PMNS_dirac_equivalence_trans {U V W : unitaryGroup (Fin 3) ℂ} :
    PMNS_dirac_equivalence U V → PMNS_dirac_equivalence V W → PMNS_dirac_equivalence U W
```

2.29 PhysLean.Particles.SuperSymmetry.SU5.ChargeSpectrum.Basic

PhysLean/Particles/SuperSymmetry/SU5/ChargeSpectrum/Basic.lean on GitHub.

def ofFinsetCard [source]

The cardinality of `ofFinset S5 S10`.

```
def ofFinsetCard (S5 S10 : Finset 𝒵) : ℕ
```

lemma ofFinset_card_eq_ofFinsetCard [source]

```
lemma ofFinset_card_eq_ofFinsetCard (S5 S10 : Finset 𝒵) :
    (ofFinset S5 S10).card = ofFinsetCard S5 S10
```

2.30 PhysLean.Relativity.LorentzGroup.Boosts.Basic

PhysLean/Relativity/LorentzGroup/Boosts/Basic.lean on GitHub.

lemma boost_zero_inl_0_inr_succ [source]

```
@[simp]
lemma boost_zero_inl_0_inr_succ {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : Fin d) :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inl 0) (Sum.inr i.succ) = 0
```

lemma boost_zero_inr_succ_inl_0 [source]

```
@[simp]
lemma boost_zero_inr_succ_inl_0{d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : Fin d) :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inr i.succ) (Sum.inl 0) = 0
```

lemma boost_zero_inl_0_inr_nat_succ [source]

```
@[simp]
lemma boost_zero_inl_0_inr_nat_succ {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : ℕ) (h : i + 1 < d + 1)
    :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inl 0) (Sum.inr ⟨i + 1, h⟩) = 0
```

lemma boost_zero_inr_nat_succ_inl_0 [source]

```
@[simp]
lemma boost_zero_inr_nat_succ_inl_0 {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : ℕ) (h : i + 1 < d + 1)
    :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inr ⟨i + 1, h⟩) (Sum.inl 0) = 0
```

lemma boost_zero_inr_0_inr_succ [source]

```
@[simp]
lemma boost_zero_inr_0_inr_succ {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : Fin d) :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inr 0) (Sum.inr i.succ) = 0
```

lemma boost_zero_inr_succ_inr_0 [source]

```
@[simp]
lemma boost_zero_inr_succ_inr_0 {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : Fin d) :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inr i.succ) (Sum.inr 0) = 0
```

lemma boost_zero_inr_0_inr_nat_succ [source]

```
@[simp]
lemma boost_zero_inr_0_inr_nat_succ {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : ℕ) (h : i + 1 < d + 1)
    :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inr 0) (Sum.inr ⟨i + 1, h⟩) = 0
```

lemma boost_zero_inr_nat_succ_inr_0 [source]

```
@[simp]
lemma boost_zero_inr_nat_succ_inr_0 {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i : ℕ) (h : i + 1 < d + 1)
    :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inr ⟨i + 1, h⟩) (Sum.inr 0) = 0
```

lemma boost_zero_inr_succ_inr_succ [source]

```
lemma boost_zero_inr_succ_inr_succ {d : ℕ} {β : ℝ} (hβ : |β| < 1) (i1 i2 : Fin d) :
    (boost (0 : Fin d.succ) β hβ).1 (Sum.inr i1.succ) (Sum.inr i2.succ) =
    if i1 = i2 then 1 else 0
```

2.31 PhysLean.Relativity.SpeedOfLight

PhysLean/Relativity/SpeedOfLight.lean on GitHub.

structure SpeedOfLight [source]

The speed of light in a vacuum. An element of this type should be thought of as the speed of light in some chosen but arbitary system of units.

```
structure SpeedOfLight where
  val : ℝ
  pos : 0 < val
```

instance Coe SpeedOfLight ℝ [source]

```
instance : Coe SpeedOfLight ℝ
```

instance One SpeedOfLight [source]

```
instance : One SpeedOfLight
```

lemma val_one [source]

```
@[simp]
lemma val_one : (1 : SpeedOfLight).val = 1
```

lemma val_pos [source]

```
@[simp]
lemma val_pos (c : SpeedOfLight) : 0 < (c : ℝ)
```

lemma val_nonneg [source]

```
@[simp]
lemma val_nonneg (c : SpeedOfLight) : 0 ≤ (c : ℝ)
```

lemma val_ne_zero [source]

```
@[simp]
lemma val_ne_zero (c : SpeedOfLight) : (c : ℝ) ≠ 0
```

2.32 PhysLean.Relativity.Tensors.Product

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

def ComponentIdx.prodIndexEquiv [source]

The equivalence between `ComponentIdx (Fin.append c c1)` and `Π (i : Fin n1 ⊕ Fin n2), Fin (S.repDim (Sum.elim c c1 i))`.

```
def ComponentIdx.prodIndexEquiv {n1 n2 : ℕ} {c : Fin n1 → C} {c1 : Fin n2 → C} :
    ComponentIdx (S := S) (Fin.append c c1) ≃
    Π (i : Fin n1 ⊕ Fin n2), Fin (S.repDim (Sum.elim c c1 i))
```

def Pure.prodIndexEquiv [source]

The equivalence between pure tensors based on a product of lists of indices, and the type `Π (i : Fin n1 ⊕ Fin n2), S.FD.obj (Discrete.mk ((Sum.elim c c1) i))`.

```
def Pure.prodIndexEquiv {n1 n2 : ℕ} {c : Fin n1 → C} {c1 : Fin n2 → C} :
    Pure S (Fin.append c c1) ≃
    Π (i : Fin n1 ⊕ Fin n2), S.FD.obj (Discrete.mk ((Sum.elim c c1) i))
```

def Tensor.prodIndexEquiv [source]

The equivalence between the type `S.F.obj (OverColor.mk (Sum.elim c c1))` and the type `S.Tensor (Fin.append c c1)`.

```
noncomputable def prodIndexEquiv {n1 n2} {c : Fin n1 → C} {c1 : Fin n2 → C} :
    S.F.obj (OverColor.mk (Sum.elim c c1)) ≃ₗ[k] S.Tensor (Fin.append c c1)
```

lemma prodIndexEquiv_symm_pure [source]

```
lemma prodIndexEquiv_symm_pure {n1 n2} {c : Fin n1 → C} {c1 : Fin n2 → C}
    (p : Pure S (Fin.append c c1)) :
    prodIndexEquiv.symm p.toTensor = PiTensorProduct.tprod k (Pure.prodIndexEquiv p)
```

2.33 PhysLean.Relativity.Tensors.RealTensor.Vector.Basic

PhysLean/Relativity/Tensors/RealTensor/Vector/Basic.lean on GitHub.

lemma apply_sum [source]

```
lemma apply_sum {d : ℕ} {ι : Type} [Fintype ι] (f : ι → Vector d) (i : Fin 1 ⊕ Fin d) :
    (∑ j, f j) i = ∑ j, f j i
```

def coordCLM [source]

The continuous linear map from a Lorentz vector to one of its coordinates.

```
def coordCLM {d : ℕ} (i : Fin 1 ⊕ Fin d) : Vector d →L[ℝ] ℝ where
```

lemma coordCLM_apply [source]

```
lemma coordCLM_apply {d : ℕ} (i : Fin 1 ⊕ Fin d) (v : Vector d) :
    coordCLM i v = v i
```

lemma fderiv_apply [source]

```
@[simp]
lemma fderiv_apply {d : ℕ} (μ : Fin 1 ⊕ Fin d) (x : Vector d) :
    fderiv ℝ (fun v : Vector d => v μ) x = coordCLM μ
```

lemma sum_basis_eq_zero_iff [source]

```
lemma sum_basis_eq_zero_iff {d : ℕ} (f : Fin 1 ⊕ Fin d → ℝ) :
    (∑ μ, f μ • basis μ) = 0 ↔ ∀ μ, f μ = 0
```

lemma sum_inl_inr_basis_eq_zero_iff [source]

```
lemma sum_inl_inr_basis_eq_zero_iff {d : ℕ} (f₀ : ℝ) (f : Fin d → ℝ) :
    f₀ • basis (Sum.inl 0) + (∑ i, f i • basis (Sum.inr i)) = 0 ↔
    f₀ = 0 ∧ ∀ i, f i = 0
```

2.34 PhysLean.SpaceAndTime.Space.Basic

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

lemma deriv_const [source]

```
@[simp]
lemma deriv_const [NormedAddCommGroup M] [NormedSpace ℝ M] (m : M) (μ : Fin d) :
    deriv μ (fun _ => m) t = 0
```

lemma direction_unit_sq_sum [source]

```
@[simp]
lemma direction_unit_sq_sum {d} (s : Direction d) :
    ∑ i : Fin d, (s.unit i) ^ 2 = 1
```

2.35 PhysLean.SpaceAndTime.Space.Distributions.Basic

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

lemma derivD_apply [source]

```
lemma derivD_apply {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (μ : Fin d) (f : (Space d) →d[ℝ] M) (ε : 𝓢(Space d, ℝ)) :
    (derivD μ f) ε = fderivD ℝ f ε (basis μ)
```

lemma gradD_toFun_eq_derivD [source]

```
lemma gradD_toFun_eq_derivD {d} (f : (Space d) →d[ℝ] ℝ) :
    (gradD f).toFun = fun ε i => derivD i f ε
```

lemma gradD_apply [source]

```
lemma gradD_apply {d} (f : (Space d) →d[ℝ] ℝ) (ε : 𝓢(Space d, ℝ)) :
    (gradD f) ε = fun i => derivD i f ε
```

lemma divD_apply_eq_sum_derivD [source]

```
lemma divD_apply_eq_sum_derivD {d}
    (f : (Space d) →d[ℝ] EuclideanSpace ℝ (Fin d)) (η : 𝓢(Space d, ℝ)) :
    divD f η = ∑ i, derivD i f η i
```

def timeDerivD [source]

The time derivative of a distribution dependent on time and space.

```
noncomputable def timeDerivD {M d} [NormedAddCommGroup M] [NormedSpace ℝ M] :
    ((Time × Space d) →d[ℝ] M) →ₗ[ℝ] (Time × Space d) →d[ℝ] M where
```

lemma timeDerivD_apply [source]

```
lemma timeDerivD_apply {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (f : (Time × Space d) →d[ℝ] M) (ε : 𝓢(Time × Space d, ℝ)) :
    (timeDerivD f) ε = fderivD ℝ f ε (1, 0)
```

def spaceDerivD [source]

The space derivative of a distribution dependent on time and space.

```
noncomputable def spaceDerivD {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (i : Fin d) : ((Time × Space d) →d[ℝ] M) →ₗ[ℝ] (Time × Space d) →d[ℝ] M where
```

lemma spaceDerivD_apply [source]

```
lemma spaceDerivD_apply {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (i : Fin d) (f : (Time × Space d) →d[ℝ] M) (ε : 𝓢(Time × Space d, ℝ)) :
    (spaceDerivD i f) ε = fderivD ℝ f ε (0, basis i)
```

def spaceGradD [source]

The spatial gradient of a distribution dependent on time and space.

```
noncomputable def spaceGradD {d} :
    ((Time × Space d) →d[ℝ] ℝ) →ₗ[ℝ] (Time × Space d) →d[ℝ] (EuclideanSpace ℝ (Fin d)) where
```

lemma spaceGradD_apply [source]

```
lemma spaceGradD_apply {d} (f : (Time × Space d) →d[ℝ] ℝ) (ε : 𝓢(Time × Space d, ℝ)) :
    spaceGradD f ε = fun i => spaceDerivD i f ε
```

def spaceDivD [source]

The spatial divergence of a distribution dependent on time and space.

```
noncomputable def spaceDivD {d} :
    ((Time × Space d) →d[ℝ] (EuclideanSpace ℝ (Fin d))) →ₗ[ℝ] (Time × Space d) →d[ℝ] ℝ where
```

lemma spaceDivD_apply_eq_sum_spaceDerivD [source]

```
lemma spaceDivD_apply_eq_sum_spaceDerivD {d}
    (f : (Time × Space d) →d[ℝ] EuclideanSpace ℝ (Fin d)) (η : 𝓢(Time ×Space d, ℝ)) :
    spaceDivD f η = ∑ i, spaceDerivD i f η i
```

def spaceCurlD [source]

The curl of a distribution dependent on time and space.

```
noncomputable def spaceCurlD : ((Time × Space 3) →d[ℝ] (EuclideanSpace ℝ (Fin 3))) →ₗ[ℝ]
    (Time × Space 3) →d[ℝ] (EuclideanSpace ℝ (Fin 3)) where
```

2.36 PhysLean.SpaceAndTime.Space.Distributions.ConstantTime

PhysLean/SpaceAndTime/Space/Distributions/ConstantTime.lean on GitHub.

lemma continuous_time_integral [source]

```
lemma continuous_time_integral {d} (η : 𝓢(Time × Space d, ℝ)) :
    Continuous (fun x : Space d => ∫ t : Time, η (t, x))
```

lemma time_integral_hasFDerivAt [source]

```
lemma time_integral_hasFDerivAt {d : ℕ} (η : 𝓢(Time × Space d.succ, ℝ)) (x₀ : Space d.succ) :
    HasFDerivAt (fun x => ∫ (t : Time), η (t, x))
      (∫ (t : Time), fderiv ℝ (fun x : Space d.succ => η (t, x)) x₀) x₀
```

lemma time_integral_differentiable [source]

```
lemma time_integral_differentiable {d : ℕ} (η : 𝓢(Time × Space d.succ, ℝ)) :
    Differentiable ℝ (fun x => ∫ (t : Time), η (t, x))
```

lemma integrable_fderiv_space [source]

```
@[fun_prop]
lemma integrable_fderiv_space {d : ℕ} (η : 𝓢(Time × Space d.succ, ℝ)) (x : Space d.succ) :
    Integrable (fun t => fderiv ℝ (fun x => η (t, x)) x) volume
```

lemma time_integral_contDiff [source]

```
lemma time_integral_contDiff {d : ℕ} (n : ℕ) (η : 𝓢(Time × Space d.succ, ℝ)) :
    ContDiff ℝ n (fun x => ∫ (t : Time), η (t, x))
```

lemma integrable_time_integral [source]

```
@[fun_prop]
lemma integrable_time_integral {d : ℕ} (η : 𝓢(Time × Space d.succ, ℝ)) (x : Space d.succ) :
    Integrable (fun t => η (t, x)) volume
```

lemma pow_mul_iteratedFDeriv_norm_le [source]

```
lemma pow_mul_iteratedFDeriv_norm_le {n m} {d : ℕ} :
    ∃ rt, ∀ (η : 𝓢(Time × Space d.succ, ℝ)), ∀ (x : Space d.succ),
    Integrable (fun x : Time => ‖((1 + ‖x‖) ^ rt)⁻¹‖) volume ∧
    ∀ t, ‖(t, x)‖ ^m * ‖iteratedFDeriv ℝ n ⇑η (t, x)‖ ≤
        (2 ^ (rt + m, n).1 *
        ((Finset.Iic (rt + m, n)).sup fun m => SchwartzMap.seminorm ℝ m.1 m.2) η) *
        ‖(1 + ‖t‖) ^ (rt)‖⁻¹
```

lemma iteratedFDeriv_norm_mul_pow_integrable [source]

```
@[fun_prop]
lemma iteratedFDeriv_norm_mul_pow_integrable {d : ℕ} (n m : ℕ) (η : 𝓢(Time × Space d.succ, ℝ))
    (x : Space d.succ) :
    Integrable (fun t => ‖(t, x)‖ ^ m * ‖iteratedFDeriv ℝ n ↿η (t, x)‖) volume
```

lemma iteratedFDeriv_norm_integrable [source]

```
@[fun_prop]
lemma iteratedFDeriv_norm_integrable {n} {d : ℕ} (η : 𝓢(Time × Space d.succ, ℝ))
    (x : Space d.succ) :
    Integrable (fun t => ‖iteratedFDeriv ℝ n ↿η (t, x)‖) volume
```

lemma iteratedFDeriv_integrable [source]

```
@[fun_prop]
lemma iteratedFDeriv_integrable {n} {d : ℕ} (η : 𝓢(Time × Space d.succ, ℝ)) (x : Space d.succ)
    :
    Integrable (fun t => iteratedFDeriv ℝ n ↿η (t, x)) volume
```

lemma time_integral_iteratedFDeriv_apply [source]

```
lemma time_integral_iteratedFDeriv_apply {d : ℕ} (n : ℕ) (η : 𝓢(Time × Space d.succ, ℝ)) :
    ∀ x, ∀ y, iteratedFDeriv ℝ n (fun x => ∫ (t : Time), η (t, x)) x y =
      ∫ (t : Time), (iteratedFDeriv ℝ n η (t, x)) (fun i => (0, y i))
```

lemma time_integral_iteratedFDeriv_eq [source]

```
lemma time_integral_iteratedFDeriv_eq {d : ℕ} (n : ℕ) (η : 𝓢(Time × Space d.succ, ℝ))
    (x : Space d.succ) :
    iteratedFDeriv ℝ n (fun x => ∫ (t : Time), η (t, x)) x =
      ((∫ (t : Time), iteratedFDeriv ℝ n η (t, x)).compContinuousLinearMap
      (fun _ => ContinuousLinearMap.prod (0 : Space d.succ →L[ℝ] Time)
      (ContinuousLinearMap.id ℝ (Space d.succ))))
```

lemma time_integral_iteratedFDeriv_norm_le [source]

```
lemma time_integral_iteratedFDeriv_norm_le {d : ℕ} (n : ℕ) (η : 𝓢(Time × Space d.succ, ℝ))
    (x : Space d.succ) :
    ‖iteratedFDeriv ℝ n (fun x => ∫ (t : Time), η (t, x)) x‖ ≤
        (∫ (t : Time), ‖iteratedFDeriv ℝ n η (t, x)‖) *
        ‖(ContinuousLinearMap.prod (0 : Space d.succ →L[ℝ] Time)
        (ContinuousLinearMap.id ℝ (Space d.succ)))‖ ^ n
```

lemma time_integral_mul_pow_iteratedFDeriv_norm_le [source]

```
lemma time_integral_mul_pow_iteratedFDeriv_norm_le {d : ℕ} (n m : ℕ) :
    ∃ rt, ∀ (η : 𝓢(Time × Space d.succ, ℝ)),∀ (x : Space d.succ),
    Integrable (fun x : Time => ‖((1 + ‖x‖) ^ rt)⁻¹‖) volume ∧
    ‖x‖ ^ m * ‖iteratedFDeriv ℝ n (fun x => ∫ (t : Time), η (t, x)) x‖ ≤
        ((∫ (t : Time), ‖((1 + ‖t‖) ^ rt)⁻¹‖) *
        ‖((0 : Space d.succ →L[ℝ] Time).prod (.id ℝ (Space d.succ)))‖ ^ n)
        * (2 ^ (rt + m, n).1 * ((Finset.Iic (rt + m, n)).sup
```

```
        fun m => SchwartzMap.seminorm ℝ m.1 m.2) η)
```

def timeIntegralSchwartz [source]

The continuous linear map taking Schwartz maps on `Time × Space d` to `space d` by integrating over time.

```
def timeIntegralSchwartz {d : ℕ} :
    𝓢(Time × Space d.succ, ℝ) →L[ℝ] 𝓢(Space d.succ, ℝ)
```

lemma timeIntegralSchwartz_apply [source]

```
lemma timeIntegralSchwartz_apply {d : ℕ} (η : 𝓢(Time × Space d.succ, ℝ)) (x : Space d.succ) :
    timeIntegralSchwartz η x = ∫ (t : Time), η (t, x)
```

def constantTime [source]

Distributions on `Time × Space d` from distributions on `Space d`. These distributions are constant in time.

```
def constantTime {M : Type} [NormedAddCommGroup M] [NormedSpace ℝ M] {d : ℕ} :
    ((Space d.succ) →d[ℝ] M) →ₗ[ℝ] (Time × Space d.succ) →d[ℝ] M where
```

lemma constantTime_apply [source]

```
lemma constantTime_apply {M : Type} [NormedAddCommGroup M] [NormedSpace ℝ M]
    {d : ℕ} (f : (Space d.succ) →d[ℝ] M)
    (η : 𝓢(Time × Space d.succ, ℝ)) :
    constantTime f η = f (timeIntegralSchwartz η)
```

lemma constantTime_spaceDerivD [source]

```
lemma constantTime_spaceDerivD {M : Type} {d : ℕ} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (i : Fin d.succ) (f : (Space d.succ) →d[ℝ] M) :
    Space.spaceDerivD i (constantTime f) = constantTime (Space.derivD i f)
```

lemma constantTime_spaceGradD [source]

```
lemma constantTime_spaceGradD {d : ℕ} (f : (Space d.succ) →d[ℝ] ℝ) :
    Space.spaceGradD (constantTime f) = constantTime (Space.gradD f)
```

lemma constantTime_spaceDivD [source]

```
lemma constantTime_spaceDivD {d : ℕ} (f : (Space d.succ) →d[ℝ] EuclideanSpace ℝ (Fin d.succ)) :
    Space.spaceDivD (constantTime f) = constantTime (Space.divD f)
```

lemma constantTime_spaceCurlD [source]

```
lemma constantTime_spaceCurlD (f : (Space 3) →d[ℝ] EuclideanSpace ℝ (Fin 3)) :
    Space.spaceCurlD (constantTime f) = constantTime (Space.curlD f)
```

lemma constantTime_timeDerivD [source]

```
@[simp]
lemma constantTime_timeDerivD {M : Type} [NormedAddCommGroup M] [NormedSpace ℝ M] {d : ℕ}
    (f : (Space d.succ) →d[ℝ] M) :
    Space.timeDerivD (constantTime f) = 0
```

2.37 PhysLean.SpaceAndTime.Space.VectorIdentities

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

lemma deriv_euclid [source]

```
lemma deriv_euclid {d ν μ} {f : Space d → EuclideanSpace ℝ (Fin n)}
    (hf : Differentiable ℝ f) (x : Space d) :
    deriv ν (fun x => f x μ) x = deriv ν (fun x => f x) x μ
```

lemma deriv_lorentz_vector [source]

```
lemma deriv_lorentz_vector {d ν μ} {f : Space d → Lorentz.Vector d}
    (hf : Differentiable ℝ f) (x : Space d) :
    deriv ν (fun x => f x μ) x = deriv ν (fun x => f x) x μ
```

lemma deriv_differentiable [source]

```
lemma deriv_differentiable {M} [NormedAddCommGroup M]
    [NormedSpace ℝ M] {d : ℕ} {f : Space d → M}
    (hf : ContDiff ℝ 2 f) (i : Fin d) :
    Differentiable ℝ (deriv i f)
```

lemma grad_neg [source]

```
lemma grad_neg (f : Space d → ℝ) :
    ∇ (- f) = - ∇ f
```

lemma grad_inner_left [source]

```
lemma grad_inner_left {d : ℕ} (x : Space d) :
    ∇ (fun y : Space d => ⟪y, x⟫_ℝ) = fun _ => x
```

lemma grad_inner_right [source]

```
lemma grad_inner_right {d : ℕ} (x : Space d) :
    ∇ (fun y : Space d => ⟪x, y⟫_ℝ) = fun _ => x
```

lemma deriv_lorentzVector [source]

```
lemma deriv_lorentzVector {d : ℕ} {f : Time → Lorentz.Vector d}
    (hf : Differentiable ℝ f) (t : Time) (i : Fin 1 ⊕ Fin d) :
    deriv (fun t => f t i) t = deriv (fun t => f t) t i
```

2.38 PhysLean.SpaceAndTime.SpaceTime.Basic

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

lemma toTimeAndSpace_symm_apply_time_space [source]

```
@[simp]
lemma toTimeAndSpace_symm_apply_time_space {d : ℕ} {c : SpeedOfLight} (x : SpaceTime d) :
    (toTimeAndSpace c).symm (x.time c, x.space) = x
```

lemma space_toTimeAndSpace_symm [source]

```
@[simp]
lemma space_toTimeAndSpace_symm {d : ℕ} {c : SpeedOfLight} (t : Time) (s : Space d) :
    ((toTimeAndSpace c).symm (t, s)).space = s
```

lemma time_toTimeAndSpace_symm [source]

```
@[simp]
lemma time_toTimeAndSpace_symm {d : ℕ} {c : SpeedOfLight} (t : Time) (s : Space d) :
    ((toTimeAndSpace c).symm (t, s)).time c = t
```

lemma toTimeAndSpace_symm_apply_inl [source]

```
@[simp]
lemma toTimeAndSpace_symm_apply_inl {d : ℕ} {c : SpeedOfLight} (t : Time) (s : Space d) :
    (toTimeAndSpace c).symm (t, s) (Sum.inl 0) = c * t
```

lemma toTimeAndSpace_symm_apply_inr [source]

```
@[simp]
lemma toTimeAndSpace_symm_apply_inr {d : ℕ} {c : SpeedOfLight} (t : Time) (x : Space d)
    (i : Fin d) :
    (toTimeAndSpace c).symm (t, x) (Sum.inr i) = x i
```

lemma toTimeAndSpace_basis_inl' [source]

```
lemma toTimeAndSpace_basis_inl' {d : ℕ} {c : SpeedOfLight} :
    toTimeAndSpace (d := d) c (Lorentz.Vector.basis (Sum.inl 0)) = (1/c.val) • (1, 0)
```

def coordCLM [source]

The continuous linear map from a point in space time to one of its coordinates.

```
def coordCLM (μ : Fin 1 ⊕ Fin d) : SpaceTime d →L[ℝ] ℝ where
```

lemma deriv_coord [source]

```
@[simp]
lemma deriv_coord {d : ℕ} (μ ν : Fin 1 ⊕ Fin d) :
    ∂_ μ (fun x => x ν) = if μ = ν then 1 else 0
```

instance MeasurableSpace (SpaceTime d) [source]

```
instance {d : ℕ} : MeasurableSpace (SpaceTime d)
```

instance BorelSpace (SpaceTime d) [source]

```
instance {d : ℕ} : BorelSpace (SpaceTime d) where
```

instance MeasureSpace (SpaceTime d) [source]

```
instance {d : ℕ} : MeasureSpace (SpaceTime d) where
```

instance (volume (α [source]

```
instance {d : ℕ} : (volume (α := SpaceTime d)).IsOpenPosMeasure
```

instance IsFiniteMeasureOnCompacts (volume (α [source]

```
instance {d : ℕ} : IsFiniteMeasureOnCompacts (volume (α := SpaceTime d))
```

instance Measure.IsAddHaarMeasure (volume (α [source]

```
instance {d : ℕ} : Measure.IsAddHaarMeasure (volume (α := SpaceTime d))
```

2.39 PhysLean.SpaceAndTime.SpaceTime.Boosts

PhysLean/SpaceAndTime/SpaceTime/Boosts.lean on GitHub.

lemma boost_zero_apply_time_space [source]

```
lemma boost_zero_apply_time_space {d : ℕ} {β : ℝ} (hβ : |β| < 1) (c : SpeedOfLight)
    (t : Time) (x : Space d.succ) :
    ((boost (0 : Fin d.succ) β hβ)⁻¹ • (SpaceTime.toTimeAndSpace c).symm (t, x)) =
    (SpaceTime.toTimeAndSpace c).symm
    (γ β * (t.val + β /c * x 0),
    fun | (0 : Fin d.succ) => γ β * (x 0 + c * β * t.val)
        | ⟨Nat.succ n, ih⟩ => x ⟨Nat.succ n, ih⟩)
```

2.40 PhysLean.SpaceAndTime.SpaceTime.Distributions

PhysLean/SpaceAndTime/SpaceTime/Distributions.lean on GitHub.

def constD [source]

The constant distribution from `SpaceTime d` to a module `M` associated with `m : M`.

```
noncomputable def constD {M } [NormedAddCommGroup M] [NormedSpace ℝ M] (d : ℕ) (m : M) :
    (SpaceTime d) →d[ℝ] M
```

def timeSliceD [source]

The time slice of a distribution on `SpaceTime d` to form a distribution on `Time × Space d`.

```
noncomputable def timeSliceD {M d} [NormedAddCommGroup M] [NormedSpace ℝ M] :
    ((SpaceTime d) →d[ℝ] M) ≃L[ℝ] ((Time × Space d) →d[ℝ] M) where
```

def derivD [source]

Given a distribution (function) `f : Space d →d[ℝ] M` the derivative of `f` in direction `μ`.

```
noncomputable def derivD {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (μ : Fin 1 ⊕ Fin d) : ((SpaceTime d) →d[ℝ] M) →ₗ[ℝ] (SpaceTime d) →d[ℝ] M where
```

lemma derivD_apply [source]

```
lemma derivD_apply {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (μ : Fin 1 ⊕ Fin d) (f : (SpaceTime d) →d[ℝ] M) (ε : 𝓢(SpaceTime d, ℝ)) :
    (derivD μ f) ε = fderivD ℝ f ε (Lorentz.Vector.basis μ)
```

lemma timeSliceD_derivD_inl [source]

```
lemma timeSliceD_derivD_inl {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (f : (SpaceTime d) →d[ℝ] M) :
    timeSliceD (derivD (Sum.inl 0) f) =
    Space.timeDerivD (timeSliceD f)
```

lemma timeSliceD_symm_derivD_inl [source]

```
lemma timeSliceD_symm_derivD_inl {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (f : (Time × Space d) →d[ℝ] M) :
    (derivD (Sum.inl 0) (timeSliceD.symm f)) =
    timeSliceD.symm (Space.timeDerivD f)
```

lemma timeSliceD_derivD_inr [source]

```
lemma timeSliceD_derivD_inr {M d} [NormedAddCommGroup M] [NormedSpace ℝ M]
    (i : Fin d) (f : (SpaceTime d) →d[ℝ] M) :
    timeSliceD (derivD (Sum.inr i) f) =
    Space.spaceDerivD i (timeSliceD f)
```

lemma timeSliceD_symm_derivD_inr [source]

```
lemma timeSliceD_symm_derivD_inr {M d} [NormedAddCommGroup M] [NormedSpace ℝ M] (i : Fin d)
    (f : (Time × Space d) →d[ℝ] M) :
    (derivD (Sum.inr i) (timeSliceD.symm f)) =
    timeSliceD.symm (Space.spaceDerivD i f)
```

2.41 PhysLean.SpaceAndTime.SpaceTime.TimeSlice

PhysLean/SpaceAndTime/SpaceTime/TimeSlice.lean on GitHub.

lemma timeSlice_contDiff [source]

```
lemma timeSlice_contDiff {d : ℕ} {M : Type} [NormedAddCommGroup M]
    [NormedSpace ℝ M]
    {n} (c : SpeedOfLight) (f : SpaceTime d → M) (h : ContDiff ℝ n f) :
    ContDiff ℝ n ↿(timeSlice c f)
```

lemma timeSlice_differentiable [source]

```
lemma timeSlice_differentiable {d : ℕ} {M : Type} [NormedAddCommGroup M]
    [NormedSpace ℝ M] (c : SpeedOfLight)
    (f : SpaceTime d → M) (h : Differentiable ℝ f) :
    Differentiable ℝ ↿(timeSlice c f)
```

2.42 PhysLean.SpaceAndTime.Time.Basic

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

lemma deriv_const [source]

```
@[simp]
lemma deriv_const [NormedAddCommGroup M] [NormedSpace ℝ M] (m : M) :
    ∂ₜ (fun _ => m) t = 0
```

lemma deriv_euclid [source]

```
lemma deriv_euclid { μ} {f : Time→ EuclideanSpace ℝ (Fin n)}
    (hf : Differentiable ℝ f) (t : Time) :
    deriv (fun t => f t μ) t = deriv (fun t => f t) t μ
```

2.43 PhysLean.StatisticalMechanics.CanonicalEnsemble.Basic

PhysLean/StatisticalMechanics/CanonicalEnsemble/Basic.lean on GitHub.

def meanSquareEnergy [source]

The mean square energy $\langle E^2 \rangle$ of the canonical ensemble at temperature T.

```
noncomputable def meanSquareEnergy (T : Temperature) : ℝ
```

def energyVariance [source]

Energy variance at temperature `T`.

```
noncomputable def energyVariance (T : Temperature) : ℝ
```

2.44 PhysLean.StatisticalMechanics.CanonicalEnsemble.Finite

PhysLean/StatisticalMechanics/CanonicalEnsemble/Finite.lean on GitHub.

lemma meanSquareEnergy_of_fintype [source]

```
lemma meanSquareEnergy_of_fintype [MeasurableSingletonClass ι] [IsFinite 𝒞] (T : Temperature) :
    𝒞.meanSquareEnergy T = ∑ i, (𝒞.energy i)^2 * 𝒞.probability T i
```

lemma energyVariance_of_fintype [source]

```
lemma energyVariance_of_fintype
    [MeasurableSingletonClass ι] [IsFinite 𝒞] [Nonempty ι] (T : Temperature) :
    𝒞.energyVariance T = (∑ i, (𝒞.energy i)^2 * 𝒞.probability T i) - (𝒞.meanEnergy T)^2
```

def mathematicalPartitionFunctionBetaReal [source]

The finite-sum partition function as a real function of the inverse temperature `b = β`, defined by `Z(b) = ∑ i exp (-b * 𝒞.energy i)`.

```
noncomputable def mathematicalPartitionFunctionBetaReal (b : ℝ) : ℝ
```

lemma mathematicalPartitionFunctionBetaReal_pos [source]

```
lemma mathematicalPartitionFunctionBetaReal_pos [Nonempty ι] (b : ℝ) :
    0 < 𝒞.mathematicalPartitionFunctionBetaReal b
```

def probabilityBetaReal [source]

For inverse temperature `b = β`, the (real-valued) Boltzmann probability of microstate `i`, given by `exp (-b * E i) / Z(b)` where `Z(b) = ∑ i exp (-b * E i)`.

```
noncomputable def probabilityBetaReal (b : ℝ) (i : ι) : ℝ
```

def meanEnergyBetaReal [source]

The mean energy as a function of inverse temperature `b = β` in the finite case, defined by `U(b) = ∑ i, E i * p_b i` with `p_b i = exp (-b * E i) / Z(b)` and `` `Z(b) = ∑ i, exp (-b * E i)` ``.

```
noncomputable def meanEnergyBetaReal (b : ℝ) : ℝ
```

lemma meanEnergy_Beta_eq_finite [source]

```
lemma meanEnergy_Beta_eq_finite [MeasurableSingletonClass ι] [IsFinite 𝒞] (b : ℝ) (hb : 0 < b)
    :
    𝒞.meanEnergyBeta b = 𝒞.meanEnergyBetaReal b
```

lemma differentiable_meanEnergyBetaReal [source]

```
lemma differentiable_meanEnergyBetaReal
    [Nonempty ι] : Differentiable ℝ 𝒞.meanEnergyBetaReal
```

lemma differentiable_mathematicalPartitionFunctionBetaReal [source]

```
lemma differentiable_mathematicalPartitionFunctionBetaReal :
    Differentiable ℝ 𝒞.mathematicalPartitionFunctionBetaReal
```

def meanEnergyNumerator [source]

The numerator in the finite-sum expression of the mean energy as a function of the inverse temperature `b = β`, namely `∑ i, E i * exp (-b * E i)` (so that `U(b) = meanEnergyNumerator b / Z(b)`).

```
noncomputable def meanEnergyNumerator (b : ℝ) : ℝ
```

lemma differentiable_meanEnergyNumerator [source]

```
lemma differentiable_meanEnergyNumerator :
    Differentiable ℝ 𝓒.meanEnergyNumerator
```

lemma deriv_mathematicalPartitionFunctionBetaReal [source]

```
lemma deriv_mathematicalPartitionFunctionBetaReal (b : ℝ) :
    deriv 𝓒.mathematicalPartitionFunctionBetaReal b = -𝓒.meanEnergyNumerator b
```

lemma deriv_meanEnergyNumerator [source]

```
lemma deriv_meanEnergyNumerator (b : ℝ) :
    deriv 𝓒.meanEnergyNumerator b =
      -∑ i, (𝓒.energy i)^2 * Real.exp (-b * 𝓒.energy i)
```

lemma deriv_meanEnergyBetaReal [source]

```
lemma deriv_meanEnergyBetaReal (b : ℝ) :
    deriv 𝓒.meanEnergyBetaReal b =
    (𝓒.meanEnergyBetaReal b)^2 - ∑ i, (𝓒.energy i)^2 * 𝓒.probabilityBetaReal b i
```

lemma derivWithin_meanEnergy_Beta_eq_neg_variance [source]

$(\partial U/\partial \beta) = -Var(E)$ for finite systems.

```
lemma derivWithin_meanEnergy_Beta_eq_neg_variance
    [MeasurableSingletonClass ι][𝓒.IsFinite] (T : Temperature) (hT_pos : 0 < T.val) :
    derivWithin 𝓒.meanEnergyBeta (Set.Ioi 0) (T.β : ℝ) = - 𝓒.energyVariance T
```

theorem fluctuation_dissipation_theorem_finite [source]

FDT for finite canonical ensembles: $C_V = Var(E) / (k_B\ T^2)$.

```
theorem fluctuation_dissipation_theorem_finite
    [MeasurableSingletonClass ι] [𝓒.IsFinite] (T : Temperature) (hT_pos : 0 < T.val) :
    𝓒.heatCapacity T = 𝓒.energyVariance T / (kB * (T.val : ℝ)^2)
```

2.45 PhysLean.StatisticalMechanics.CanonicalEnsemble.Lemmas

PhysLean/StatisticalMechanics/CanonicalEnsemble/Lemmas.lean on GitHub.

theorem energyVariance_eq_meanSquareEnergy_sub_meanEnergy_sq [source]

The identity $Var(E) = \langle E^2 \rangle - \langle E \rangle^2$.

```
theorem energyVariance_eq_meanSquareEnergy_sub_meanEnergy_sq
    (𝒞 : CanonicalEnsemble ι) (T : Temperature) [IsProbabilityMeasure (𝒞.μProd T)]
    (hE_int : Integrable 𝒞.energy (𝒞.μProd T))
    (hE2_int : Integrable (fun i => (𝒞.energy i)^2) (𝒞.μProd T)) :
    𝒞.energyVariance T = 𝒞.meanSquareEnergy T - (𝒞.meanEnergy T)^2
```

def meanEnergy_T [source]

The mean energy as a function of the real-valued temperature t.

```
noncomputable def meanEnergy_T (𝒞 : CanonicalEnsemble ι) (t : ℝ) : ℝ
```

def meanEnergyBeta [source]

The mean energy as a function of the real-valued inverse temperature b.

```
noncomputable def meanEnergyBeta (𝒞 : CanonicalEnsemble ι) (b : ℝ) : ℝ
```

def heatCapacity [source]

The heat capacity (at constant volume) C_V = ∂U/∂T (as a derivWithin on T > 0).

```
noncomputable def heatCapacity (𝒞 : CanonicalEnsemble ι) (T : Temperature) : ℝ
```

lemma heatCapacity_eq_deriv_meanEnergyBeta [source]

*Relates C_V = dU/dT to dU/dβ. C_V = dU/dβ * (-1/(kB T²)).*

```
lemma heatCapacity_eq_deriv_meanEnergyBeta
    (𝒞 : CanonicalEnsemble ι) (T : Temperature) (hT_pos : 0 < T.val)
    (hU_deriv :
      HasDerivWithinAt (𝒞.meanEnergyBeta)
        (derivWithin (𝒞.meanEnergyBeta) (Set.Ioi 0) (T.β : ℝ))
        (Set.Ioi 0) (T.β : ℝ)) :
    𝒞.heatCapacity T
      = (derivWithin (𝒞.meanEnergyBeta) (Set.Ioi 0) (T.β : ℝ))
        * (-1 / (kB * (T.val : ℝ)^2))
```

theorem fluctuation_dissipation_energy_parametric [source]

Parametric FDT: C_V = Var(E)/(kB T²), assuming Var(E) = - dU/dβ.

```
theorem fluctuation_dissipation_energy_parametric
    (𝒞 : CanonicalEnsemble ι) (T : Temperature) (hT_pos : 0 < T.val)
    (h_Var_eq_neg_dUdβ :
      𝒞.energyVariance T = - derivWithin (𝒞.meanEnergyBeta) (Set.Ioi 0) (T.β : ℝ))
    (hU_deriv :
      DifferentiableWithinAt ℝ (𝒞.meanEnergyBeta) (Set.Ioi 0) (T.β : ℝ)) :
    𝒞.heatCapacity T = 𝒞.energyVariance T / (kB * (T.val : ℝ)^2)
```

2.46 PhysLean.StringTheory.FTheory.SU5.Charges.Viable

PhysLean/StringTheory/FTheory/SU5/Charges/Viable.lean on GitHub.

lemma viableCharges_mem_ofFinset [source]

```
lemma viableCharges_mem_ofFinset (I : CodimensionOneConfig) :
    ∀ x ∈ (viableCharges I), x ∈ ofFinset I.allowedBarFiveCharges I.allowedTenCharges
```

lemma mem_viableCharges_iff' [source]

```
lemma mem_viableCharges_iff' {I} {x : ChargeSpectrum} :
    x ∈ viableCharges I ↔
    x ∈ ofFinset I.allowedBarFiveCharges I.allowedTenCharges ∧
    AllowsTerm x topYukawa ∧
    ¬ IsPhenoConstrained x ∧ ¬ YukawaGeneratesDangerousAtLevel x 1 ∧ IsComplete x
```

2.47 PhysLean.StringTheory.FTheory.SU5.Fluxes.Basic

PhysLean/StringTheory/FTheory/SU5/Fluxes/Basic.lean on GitHub.

structure Fluxes [source]

The data of the fluxes carried by a matter field.

```
structure Fluxes where
  M : ℤ
  N : ℤ
```

instance Repr Fluxes [source]

```
instance : Repr Fluxes where
```

lemma ext_iff [source]

```
lemma ext_iff {f1 f2 : Fluxes} : f1 = f2 ↔ f1.M = f2.M ∧ f1.N = f2.N
```

instance Zero Fluxes [source]

```
instance : Zero Fluxes
```

lemma zero_M [source]

```
@[simp]
lemma zero_M : (0 : Fluxes).M = 0
```

lemma zero_N [source]

```
@[simp]
lemma zero_N : (0 : Fluxes).N = 0
```

instance Add Fluxes [source]

```
instance : Add Fluxes where
```

lemma add_M [source]

```
@[simp]
lemma add_M (f1 f2 : Fluxes) : (f1 + f2).M = f1.M + f2.M
```

lemma add_N [source]

```
@[simp]
lemma add_N (f1 f2 : Fluxes) : (f1 + f2).N = f1.N + f2.N
```

instance AddCommMonoid Fluxes [source]

```
instance : AddCommMonoid Fluxes where
```

2.48 PhysLean.StringTheory.FTheory.SU5.Fluxes.NoExotics.Completeness

PhysLean/StringTheory/FTheory/SU5/Fluxes/NoExotics/Completeness.lean on GitHub.

lemma FluxesFive.subset_le_mem_of_card_eq_succ [source]

```
lemma subset_le_mem_of_card_eq_succ {n : ℕ} {F : FluxesFive} (hF : F.NoExotics) (hnZ : F.
    HasNoZero)
    (S : Multiset Fluxes) (hcard : S.card = n.succ) (hS : S ≤ F) {Y X : Finset (Multiset Fluxes
    )}
    (hY : ∀ (S : Multiset Fluxes), S.card = n → S ≤ F → S ∈ Y)
    (hX : ∀ a ∈ ({⟨0, 1⟩, ⟨0, 2⟩, ⟨0, 3⟩, ⟨1, -1⟩, ⟨1, 0⟩, ⟨1, 1⟩, ⟨1, 2⟩,
      ⟨2, -2⟩, ⟨2, -1⟩, ⟨2, 0⟩, ⟨2, 1⟩, ⟨3, -3⟩, ⟨3, -2⟩, ⟨3, -1⟩, ⟨3, 0⟩} : Finset Fluxes),
      ∀ y ∈ Y,
      (Multiset.map (fun x => x.M + x.N) (a ::ₘ y)).sum ≤ 3 →
      (Multiset.map (fun x => x.M) (a ::ₘ y)).sum ≤ 3 →
      a ::ₘ y ∈ X) :
    S ∈ X
```

def FluxesFive.noExoticsSubsets [source]

The allowed subsets of a `FluxesFive` which has no exotics or zeros.

```
def noExoticsSubsets : (n : ℕ) → Finset (Multiset Fluxes)
  | 0 => {{}}
  | 1 => {{⟨0, 1⟩}, {⟨0, 2⟩}, {⟨0, 3⟩}, {⟨1, -1⟩}, {⟨1, 0⟩}, {⟨1, 1⟩}, {⟨1, 2⟩},
      {⟨2, -2⟩}, {⟨2, -1⟩}, {⟨2, 0⟩}, {⟨2, 1⟩}, {⟨3, -3⟩}, {⟨3, -2⟩}, {⟨3, -1⟩}, {⟨3, 0⟩}}
  | 2 => {{⟨0, 1⟩, ⟨0, 1⟩}, {⟨0, 1⟩, ⟨1, -1⟩}, {⟨0, 1⟩, ⟨1, 0⟩}, {⟨0, 1⟩, ⟨0, 2⟩},
      {⟨0, 1⟩, ⟨1, 1⟩}, {⟨0, 1⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨2, -1⟩}, {⟨0, 1⟩, ⟨2, 0⟩},
      {⟨0, 1⟩, ⟨3, -3⟩}, {⟨0, 1⟩, ⟨3, -2⟩}, {⟨0, 1⟩, ⟨3, -1⟩},
      {⟨0, 2⟩, ⟨1, -1⟩}, {⟨0, 2⟩, ⟨1, 0⟩}, {⟨0, 2⟩, ⟨2, -2⟩}, {⟨0, 2⟩, ⟨2, -1⟩},
      {⟨0, 2⟩, ⟨3, -3⟩}, {⟨0, 2⟩, ⟨3, -2⟩}, {⟨0, 3⟩, ⟨1, -1⟩}, {⟨0, 3⟩, ⟨2, -2⟩},
      {⟨0, 3⟩, ⟨3, -3⟩}, {⟨1, -1⟩, ⟨1, -1⟩}, {⟨1, -1⟩, ⟨1, 0⟩}, {⟨1, -1⟩, ⟨1, 1⟩},
      {⟨1, -1⟩, ⟨1, 2⟩}, {⟨1, -1⟩, ⟨2, -2⟩}, {⟨1, -1⟩, ⟨2, -1⟩}, {⟨1, -1⟩, ⟨2, 0⟩},
      {⟨1, 0⟩, ⟨1, 0⟩}, {⟨1, -1⟩, ⟨2, 1⟩}, {⟨1, 0⟩, ⟨1, 1⟩}, {⟨1, 0⟩, ⟨2, -2⟩},
      {⟨1, 0⟩, ⟨2, -1⟩}, {⟨1, 0⟩, ⟨2, 0⟩}, {⟨1, 1⟩, ⟨2, -2⟩}, {⟨1, 1⟩, ⟨2, -1⟩},
      {⟨1, 2⟩, ⟨2, -2⟩}}
  | 3 => {{⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩}, {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, 0⟩}, {⟨0, 1⟩, ⟨0, 1⟩, ⟨2, -2⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨2, -1⟩}, {⟨0, 1⟩, ⟨0, 1⟩, ⟨3, -3⟩}, {⟨0, 1⟩, ⟨0, 1⟩, ⟨3, -2⟩},
    {⟨0, 1⟩, ⟨0, 2⟩, ⟨1, -1⟩}, {⟨0, 1⟩, ⟨0, 2⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨0, 2⟩, ⟨3, -3⟩},
    {⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩}, {⟨0, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩}, {⟨0, 1⟩, ⟨1, -1⟩, ⟨1, 1⟩},
```

```
    {⟨0, 1⟩, ⟨1, -1⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨1, -1⟩, ⟨2, -1⟩}, {⟨0, 1⟩, ⟨1, -1⟩, ⟨2, 0⟩},
    {⟨0, 1⟩, ⟨1, 0⟩, ⟨1, 0⟩}, {⟨0, 1⟩, ⟨1, 0⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨1, 0⟩, ⟨2, -1⟩},
    {⟨0, 1⟩, ⟨1, 1⟩, ⟨2, -2⟩}, {⟨0, 2⟩, ⟨1, -1⟩, ⟨1, -1⟩}, {⟨0, 2⟩, ⟨1, -1⟩, ⟨1, 0⟩},
    {⟨0, 2⟩, ⟨1, -1⟩, ⟨2, -2⟩}, {⟨0, 2⟩, ⟨1, -1⟩, ⟨2, -1⟩}, {⟨0, 2⟩, ⟨1, 0⟩, ⟨2, -2⟩},
    {⟨0, 3⟩, ⟨1, -1⟩, ⟨1, -1⟩}, {⟨0, 3⟩, ⟨1, -1⟩, ⟨2, -2⟩}, {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩},
    {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, 0⟩}, {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, 1⟩}, {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, 2⟩},
    {⟨1, -1⟩, ⟨1, 0⟩, ⟨1, 0⟩}, {⟨1, -1⟩, ⟨1, 0⟩, ⟨1, 1⟩}, {⟨1, 0⟩, ⟨1, 0⟩, ⟨1, 0⟩}}
  | 4 => {{⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨3, -3⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩}, {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨2, -1⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, 0⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨0, 2⟩, ⟨1, -1⟩, ⟨1, -1⟩},
    {⟨0, 1⟩, ⟨0, 2⟩, ⟨1, -1⟩, ⟨2, -2⟩}, {⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩},
    {⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, 0⟩}, {⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, 1⟩},
    {⟨0, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩, ⟨1, 0⟩}, {⟨0, 2⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩},
    {⟨0, 2⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, 0⟩}, {⟨0, 3⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩}}
  | 5 => {{⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨2, -2⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩},
    {⟨0, 1⟩, ⟨0, 1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, 0⟩},
    {⟨0, 1⟩, ⟨0, 2⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩}}
```

lemma subset_of_fluxesFive_mem_noExoticsSubsets_of_noExotics [source]

```
lemma subset_of_fluxesFive_mem_noExoticsSubsets_of_noExotics {F : FluxesFive} (hF : F.NoExotics
    )
    (hnZ : F.HasNoZero) (S : Multiset Fluxes) (hS : S ≤ F) :
    S ∈ noExoticsSubsets S.card
```

lemma FluxesTen.subset_le_mem_of_card_eq_succ [source]

```
lemma subset_le_mem_of_card_eq_succ {n : ℕ} {F : FluxesTen} (hF : F.NoExotics) (hnZ : F.
    HasNoZero)
    (S : Multiset Fluxes) (hcard : S.card = n.succ) (hS : S ≤ F) {Y X : Finset (Multiset Fluxes
    )}
    (hY : ∀ (S : Multiset Fluxes), S.card = n → S ≤ F → S ∈ Y)
    (hX : ∀ a ∈ ({⟨1, -1⟩, ⟨1, 0⟩, ⟨1, 1⟩, ⟨2, -1⟩, ⟨2, 0⟩, ⟨2, 1⟩, ⟨3, 0⟩} : Finset Fluxes),
      ∀ y ∈ Y,
      (Multiset.map (fun x => x.M) (a ::ₘ y)).sum ≤ 3 →
      (Multiset.map (fun x => x.M - x.N) (a ::ₘ y)).sum ≤ 3 →
      (Multiset.map (fun x => x.M + x.N) (a ::ₘ y)).sum ≤ 3 →
      a ::ₘ y ∈ X) :
    S ∈ X
```

def FluxesTen.noExoticsSubsets [source]

The allowed subsets of a `FluxesTen` which has no exotics or zeros.

```
def noExoticsSubsets : (n : ℕ) → Finset (Multiset Fluxes)
  | 0 => {{}}
  | 1 => {{⟨1, -1⟩}, {⟨1, 0⟩}, {⟨1, 1⟩}, {⟨2, -1⟩}, {⟨2, 0⟩}, {⟨2, 1⟩}, {⟨3, 0⟩}}
  | 2 => {{⟨1, -1⟩, ⟨1, 0⟩}, {⟨1, -1⟩, ⟨1, 1⟩}, {⟨1, -1⟩, ⟨2, 1⟩},
      {⟨1, 0⟩, ⟨1, 0⟩}, {⟨1, 0⟩, ⟨1, 1⟩}, {⟨1, 0⟩, ⟨2, 0⟩}, {⟨1, 1⟩, ⟨2, -1⟩}}
  | 3 => {{⟨1, -1⟩, ⟨1, 0⟩, ⟨1, 1⟩}, {⟨1, 0⟩, ⟨1, 0⟩, ⟨1, 0⟩}}
  | _ + 4 => ∅
```

lemma subset_of_fluxesTen_mem_noExoticsSubsets_of_noExotics [source]

```
lemma subset_of_fluxesTen_mem_noExoticsSubsets_of_noExotics {F : FluxesTen} (hF : F.NoExotics)
    (hnZ : F.HasNoZero) (S : Multiset Fluxes) (hS : S ≤ F) :
    S ∈ noExoticsSubsets S.card
```

2.49 PhysLean.StringTheory.FTheory.SU5.Quanta.Basic

PhysLean/StringTheory/FTheory/SU5/Quanta/Basic.lean on GitHub.

instance Repr (Quanta $\mathcal{Z}$) [source]

```
unsafe instance [Repr 𝓩] : Repr (Quanta 𝓩) where
```

lemma ext [source]

```
@[ext]
lemma ext {𝓩 : Type} {x y : Quanta 𝓩} (h1 : x.qHd = y.qHd) (h2 : x.qHu = y.qHu)
    (h3 : x.F = y.F) (h4 : x.T = y.T) : x = y
```

lemma toCharges_qHd [source]

```
lemma toCharges_qHd [DecidableEq 𝓩] (x : Quanta 𝓩) : (toCharges x).qHd = x.qHd
```

lemma toCharges_qHu [source]

```
lemma toCharges_qHu [DecidableEq 𝓩] (x : Quanta 𝓩) : (toCharges x).qHu = x.qHu
```

def liftCharge [source]

Lifting a charge spectrum to quanta which do not have exotics and which have no zero flux.

```
def liftCharge [DecidableEq 𝓩] (c : ChargeSpectrum 𝓩) : Multiset (Quanta 𝓩)
```

lemma mem_liftCharge_iff [source]

```
lemma mem_liftCharge_iff [DecidableEq 𝓩] {c : ChargeSpectrum 𝓩}
    {x : Quanta 𝓩} :
    x ∈ liftCharge c ↔ x.qHd = c.qHd ∧ x.qHu = c.qHu ∧
    x.F ∈ FiveQuanta.liftCharge c.Q5 ∧ x.T ∈ TenQuanta.liftCharge c.Q10
```

lemma toCharges_of_mem_liftCharge [source]

```
lemma toCharges_of_mem_liftCharge [DecidableEq 𝓩] {c : ChargeSpectrum 𝓩}
    {x : Quanta 𝓩} (h : x ∈ liftCharge c) :
    x.toCharges = c
```

def HdAnomalyCoefficient [source]

The pair of anomaly cancellation coefficients associated with the `Hd` particle.

```
def HdAnomalyCoefficient [CommRing ℤ] (qHd : Option ℤ) : ℤ × ℤ
```

lemma HdAnomalyCoefficient_map [source]

```
@[simp]
lemma HdAnomalyCoefficient_map {ℤ ℤ1 : Type} [CommRing ℤ] [CommRing ℤ1]
    (f : ℤ →+* ℤ1) (qHd : Option ℤ) :
    HdAnomalyCoefficient (qHd.map f) = (f.prodMap f) (HdAnomalyCoefficient qHd)
```

def HuAnomalyCoefficient [source]

The pair of anomaly cancellation coefficients associated with the `Hu` particle.

```
def HuAnomalyCoefficient [CommRing ℤ] (qHu : Option ℤ) : ℤ × ℤ
```

lemma HuAnomalyCoefficient_map [source]

```
@[simp]
lemma HuAnomalyCoefficient_map {ℤ ℤ1 : Type} [CommRing ℤ] [CommRing ℤ1]
    (f : ℤ →+* ℤ1) (qHu : Option ℤ) :
    HuAnomalyCoefficient (qHu.map f) = (f.prodMap f) (HuAnomalyCoefficient qHu)
```

def LinearAnomalyCancellation [source]

The linear anomaly cancellation condition, corresponding to $\sum_i q_i N_i + \sum_a q_a N_a = 0$ where the first sum is over all 5-bar representations and the second is over all 10d representations.

```
def LinearAnomalyCancellation [CommRing ℤ] (Q : Quanta ℤ) : Prop
```

def QuarticAnomalyCancellation [source]

The quartic anomaly cancellation condition, corresponding to $\sum_i q_i^2 N_i + 3 * \sum_a q_a^2 N_a = 0$ where the first sum is over all 5-bar representations and the second is over all 10d representations.

```
def QuarticAnomalyCancellation [CommRing ℤ] (Q : Quanta ℤ) :
    Prop
```

instance Decidable Q.LinearAnomalyCancellation [source]

```
instance [CommRing ℤ] [DecidableEq ℤ] (Q : Quanta ℤ) : Decidable Q.LinearAnomalyCancellation
```

instance Decidable Q.QuarticAnomalyCancellation [source]

```
instance [CommRing ℤ] [DecidableEq ℤ] (Q : Quanta ℤ) : Decidable Q.QuarticAnomalyCancellation
```

2.50 PhysLean.StringTheory.FTheory.SU5.Quanta.FiveQuanta

PhysLean/StringTheory/FTheory/SU5/Quanta/FiveQuanta.lean on GitHub.

def toChargeMap [source]

The map which takes a charge to the overall flux it corresponds to in a `FiveQuanta`.

```
def toChargeMap [DecidableEq ℤ] (x : FiveQuanta ℤ) : ℤ → Fluxes
```

lemma toChargeMap_of_not_mem [source]

```
lemma toChargeMap_of_not_mem [DecidableEq ℤ] (x : FiveQuanta ℤ) {z : ℤ} (h : z ∉ x.toCharges)
     :
    x.toChargeMap z = 0
```

lemma reduce_toChargeMap_eq [source]

```
lemma reduce_toChargeMap_eq (x : FiveQuanta ℤ) :
    x.reduce.toChargeMap = x.toChargeMap
```

lemma mem_powerset_sum_of_mem_reduce_toFluxesFive [source]

```
lemma mem_powerset_sum_of_mem_reduce_toFluxesFive {F : FiveQuanta ℤ}
    {f : Fluxes} (hf : f ∈ F.reduce.toFluxesFive) :
    f ∈ (Multiset.powerset F.toFluxesFive).map fun s => s.sum
```

lemma mem_powerset_sum_of_mem_reduce_toFluxesFive_filter [source]

```
lemma mem_powerset_sum_of_mem_reduce_toFluxesFive_filter {F : FiveQuanta ℤ}
    {f : Fluxes} (hf : f ∈ F.reduce.toFluxesFive) :
    f ∈ (F.toFluxesFive.powerset.filter fun s => s ≠ ∅).map fun s => s.sum
```

lemma reduce_numChiralL_of_mem_elemsNoExotics [source]

```
lemma reduce_numChiralL_of_mem_elemsNoExotics {F : FiveQuanta ℤ}
    (hx : F.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    F.reduce.toFluxesFive.numChiralL = 3
```

lemma reduce_numAntiChiralL_of_mem_elemsNoExotics [source]

```
lemma reduce_numAntiChiralL_of_mem_elemsNoExotics {F : FiveQuanta ℤ}
    (hx : F.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    F.reduce.toFluxesFive.numAntiChiralL = 0
```

lemma reduce_numChiralD_of_mem_elemsNoExotics [source]

```
lemma reduce_numChiralD_of_mem_elemsNoExotics {F : FiveQuanta ℤ}
    (hx : F.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    F.reduce.toFluxesFive.numChiralD = 3
```

lemma reduce_numAntiChiralD_of_mem_elemsNoExotics [source]

```
lemma reduce_numAntiChiralD_of_mem_elemsNoExotics {F : FiveQuanta ℤ}
    (hx : F.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    F.reduce.toFluxesFive.numAntiChiralD = 0
```

lemma reduce_noExotics_of_mem_elemsNoExotics [source]

```
lemma reduce_noExotics_of_mem_elemsNoExotics {F : FiveQuanta ℤ}
    (hx : F.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    F.reduce.toFluxesFive.NoExotics
```

lemma reduce_mem_elemsNoExotics [source]

```
lemma reduce_mem_elemsNoExotics {F : FiveQuanta ℤ}
    (hx : F.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    F.reduce.toFluxesFive ∈ FluxesFive.elemsNoExotics
```

def decomposeFluxes [source]

The decomposition of a flux into `⟨1, -1⟩` and `⟨0, 1⟩`.

```
def decomposeFluxes (f : Fluxes) : Multiset Fluxes
```

lemma decomposeFluxes_sum_of_noExotics [source]

```
lemma decomposeFluxes_sum_of_noExotics (f : Fluxes) (hf : ∃ F ∈ FluxesFive.elemsNoExotics, f ∈
    F) :
    (decomposeFluxes f).sum = f
```

def decompose [source]

The decomposition of a `FiveQuanta` into a `FiveQuanta` which has the same `reduce` by has fluxes `⟨1, -1⟩` and `⟨0,1⟩` only.

```
def decompose (x : FiveQuanta ℤ) : FiveQuanta ℤ
```

lemma decompose_add [source]

```
lemma decompose_add (x y : FiveQuanta ℤ) :
    (x + y).decompose = x.decompose + y.decompose
```

lemma decompose_filter_charge [source]

```
lemma decompose_filter_charge [DecidableEq ℤ] (x : FiveQuanta ℤ) (q : ℤ) :
    (x.decompose).filter (fun p => p.1 = q) =
    decompose (x.filter (fun p => p.1 = q))
```

lemma decompose_toChargeMap [source]

```
lemma decompose_toChargeMap [DecidableEq ℤ] (x : FiveQuanta ℤ)
    (hx : x.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    x.decompose.toChargeMap = x.toChargeMap
```

lemma decompose_toCharges_dedup [source]

```
lemma decompose_toCharges_dedup [DecidableEq ℤ] (x : FiveQuanta ℤ)
    (hx : x.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    x.decompose.toCharges.dedup = x.toCharges.dedup
```

lemma decompose_reduce [source]

```
lemma decompose_reduce (x : FiveQuanta ℤ) [DecidableEq ℤ]
    (hx : x.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    x.decompose.reduce = x.reduce
```

lemma decompose_toFluxesFive [source]

```
lemma decompose_toFluxesFive (x : FiveQuanta ℤ)
    (hx : x.toFluxesFive ∈ FluxesFive.elemsNoExotics) :
    x.decompose.toFluxesFive = {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩}
```

def liftCharge [source]

Given a finite set of charges `c` the `FiveQuanta` which do not have exotics, duplicate charges or zero fluxes, which map down to `c`.

```
def liftCharge (c : Finset ℤ) : Multiset (FiveQuanta ℤ)
```

lemma toCharges_toFinset_of_mem_liftCharge [source]

```
lemma toCharges_toFinset_of_mem_liftCharge (c : Finset ℤ) {x : FiveQuanta ℤ}
    (h : x ∈ liftCharge c) : x.toCharges.toFinset = c
```

lemma toCharges_nodup_of_mem_liftCharge [source]

```
lemma toCharges_nodup_of_mem_liftCharge (c : Finset ℤ) {x : FiveQuanta ℤ}
    (h : x ∈ liftCharge c) : x.toCharges.Nodup
```

lemma exists_toCharges_toFluxesFive_of_mem_liftCharge [source]

```
lemma exists_toCharges_toFluxesFive_of_mem_liftCharge (c : Finset ℤ) {x : FiveQuanta ℤ}
    (h : x ∈ liftCharge c) :
    ∃ a : FiveQuanta ℤ, a.reduce = x ∧ a.toCharges.toFinset = c ∧ a.toFluxesFive =
      {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩}
```

lemma mem_liftCharge_of_exists_toCharges_toFluxesFive [source]

```
lemma mem_liftCharge_of_exists_toCharges_toFluxesFive (c : Finset ℤ) {x : FiveQuanta ℤ}
    (h : ∃ a : FiveQuanta ℤ, a.reduce = x ∧ a.toCharges.toFinset = c ∧ a.toFluxesFive =
      {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩}) :
    x ∈ liftCharge c
```

lemma mem_liftCharge_iff_exists [source]

```
lemma mem_liftCharge_iff_exists (c : Finset ℤ) {x : FiveQuanta ℤ} :
    x ∈ liftCharge c ↔ ∃ a : FiveQuanta ℤ, a.reduce = x ∧
      a.toCharges.toFinset = c ∧ a.toFluxesFive =
      {⟨1, -1⟩, ⟨1, -1⟩, ⟨1, -1⟩, ⟨0, 1⟩, ⟨0, 1⟩, ⟨0, 1⟩}
```

lemma hasNoZero_of_mem_liftCharge [source]

```
lemma hasNoZero_of_mem_liftCharge (c : Finset ℤ) {x : FiveQuanta ℤ}
    (h : x ∈ liftCharge c) : x.toFluxesFive.HasNoZero
```

lemma noExotics_of_mem_liftCharge [source]

```
lemma noExotics_of_mem_liftCharge (c : Finset ℤ) (F : FiveQuanta ℤ)
    (h : F ∈ liftCharge c) :
    F.toFluxesFive.NoExotics
```

lemma mem_liftCharge_of_mem_noExotics_hasNoZero [source]

```
lemma mem_liftCharge_of_mem_noExotics_hasNoZero (c : Finset ℤ) {x : FiveQuanta ℤ}
    (h1 : x.toFluxesFive.NoExotics) (h2 : x.toFluxesFive.HasNoZero)
    (h3 : x.toCharges.toFinset = c) (h4 : x.toCharges.Nodup) :
    x ∈ liftCharge c
```

lemma mem_liftCharge_iff [source]

```
lemma mem_liftCharge_iff (c : Finset ℤ) (x : FiveQuanta ℤ) :
    x ∈ liftCharge c ↔ x.toFluxesFive ∈ FluxesFive.elemsNoExotics
      ∧ x.toCharges.toFinset = c ∧ x.toCharges.Nodup
```

lemma map_liftCharge [source]

```
lemma map_liftCharge {ℤ ℤ1 : Type}[DecidableEq ℤ] [DecidableEq ℤ1] [CommRing ℤ] [CommRing
    ℤ1]
    (f : ℤ →+* ℤ1) (c : Finset ℤ) (F : FiveQuanta ℤ) (h : F ∈ liftCharge c) :
    FiveQuanta.reduce (F.map fun y => (f y.1, y.2)) ∈ liftCharge (c.image f)
```

def anomalyCoefficient [source]

The anomaly coefficient of a `FiveQuanta` is given by the pair of integers: `(∑ᵢ qᵢ Nᵢ, ∑ᵢ qᵢ² Nᵢ)`.

The first components is for the mixed U(1)-MSSM, see equation (22) of arXiv:1401.5084. The second component is for the mixed U(1)Y-U(1)-U(1) gauge anomaly, see equation (23) of arXiv:1401.5084.

```
def anomalyCoefficient (F : FiveQuanta ℤ) : ℤ × ℤ
```

lemma anomalyCoefficient_of_map [source]

```
@[simp]
lemma anomalyCoefficient_of_map {ℤ ℤ1 : Type} [CommRing ℤ] [CommRing ℤ1]
    (f : ℤ →+* ℤ1) (F : FiveQuanta ℤ) :
    FiveQuanta.anomalyCoefficient (F.map fun y => (f y.1, y.2) : FiveQuanta ℤ1) =
    (f.prodMap f) F.anomalyCoefficient
```

lemma anomalyCoefficient_of_reduce [source]

```
lemma anomalyCoefficient_of_reduce (F : FiveQuanta ℤ) [DecidableEq ℤ] :
    F.reduce.anomalyCoefficient = F.anomalyCoefficient
```

2.51 PhysLean.StringTheory.FTheory.SU5.Quanta.IsViable

PhysLean/StringTheory/FTheory/SU5/Quanta/IsViable.lean on GitHub.

lemma isViable_iff_def [source]

```
lemma isViable_iff_def (x : Quanta) : IsViable x ↔
    x.toCharges.IsComplete ∧
    ¬ x.toCharges.IsPhenoConstrained ∧
    ¬ x.toCharges.YukawaGeneratesDangerousAtLevel 1 ∧
    (∃ I : CodimensionOneConfig, x.toCharges ∈
      ofFinset I.allowedBarFiveCharges I.allowedTenCharges) ∧
    x.F.toCharges.Nodup ∧
    x.T.toCharges.Nodup ∧
    x.toCharges.AllowsTerm topYukawa ∧
    x.F.toFluxesFive.NoExotics ∧
    x.F.toFluxesFive.HasNoZero ∧
    x.T.toFluxesTen.NoExotics ∧
    x.T.toFluxesTen.HasNoZero ∧
    x.LinearAnomalyCancellation
```

lemma isViable_iff_charges_mem_viableCharges [source]

```
lemma isViable_iff_charges_mem_viableCharges (x : Quanta) :
    IsViable x ↔
```

lemma isViable_iff_charges_mem_viableCharges_mem_liftCharges [source]

```
lemma isViable_iff_charges_mem_viableCharges_mem_liftCharges (x : Quanta) :
    IsViable x ↔ (∃ I, x.toCharges ∈ viableCharges I) ∧
      x ∈ Quanta.liftCharge x.toCharges ∧
      x.LinearAnomalyCancellation
```

lemma isViable_iff_filter [source]

```
lemma isViable_iff_filter (x : Quanta) :
    IsViable x ↔ (∃ I, x.toCharges ∈ (viableCharges I).filter IsAnomalyFree) ∧
      x ∈ Quanta.liftCharge x.toCharges
      ∧ x.LinearAnomalyCancellation
```

lemma quarticAnomalyCancellation_iff_mem_of_isViable [source]

```
lemma quarticAnomalyCancellation_iff_mem_of_isViable (x : Quanta) (h : IsViable x) :
    x.QuarticAnomalyCancellation ↔ x ∈
    ({⟨some 2, some (-2), {(-1, ⟨1, 2⟩), (1, ⟨2, -2⟩)}, {(-1, ⟨3, 0⟩)}⟩,
    ⟨some 2, some (-2), {(-1, ⟨0, 2⟩), (1, ⟨3, -2⟩)}, {(-1, ⟨3, 0⟩)}⟩,
    ⟨some (-2), some 2, {(-1, ⟨2, -2⟩), (1, ⟨1, 2⟩)}, {(1, ⟨3, 0⟩)}⟩,
    ⟨some (-2), some 2, {(-1, ⟨3, -2⟩), (1, ⟨0, 2⟩)}, {(1, ⟨3, 0⟩)}⟩,
    ⟨some 6, some (-14), {(-9, ⟨1, 2⟩), (1, ⟨2, -2⟩)}, {(-7, ⟨3, 0⟩)}⟩,
    ⟨some 6, some (-14), {(-9, ⟨0, 2⟩), (1, ⟨3, -2⟩)}, {(-7, ⟨3, 0⟩)}⟩} : Finset Quanta)
```

lemma map_to_Z2_of_isViable [source]

```
lemma map_to_Z2_of_isViable (x : Quanta ℤ) (h : IsViable x) :
    x.toCharges.map (Int.castAddHom (ZMod 2)) ∈
    ({⟨some 0, some 0, {1}, {1}⟩, ⟨some 0, some 0, {0,1}, {1}⟩, ⟨some 0, some 0, {0,1}, {0}⟩} :
    Finset (ChargeSpectrum (ZMod 2)))
```

2.52 PhysLean.StringTheory.FTheory.SU5.Quanta.TenQuanta

PhysLean/StringTheory/FTheory/SU5/Quanta/TenQuanta.lean on GitHub.

def toChargeMap [source]

The map which takes a charge to the overall flux it corresponds to in a `TenQuanta`.

```
def toChargeMap [DecidableEq 𝒵] (x : TenQuanta 𝒵) : 𝒵 → Fluxes
```

lemma toChargeMap_of_not_mem [source]

```
lemma toChargeMap_of_not_mem [DecidableEq 𝒵] (x : TenQuanta 𝒵) {z : 𝒵} (h : z ∉ x.toCharges)
    :
    x.toChargeMap z = 0
```

lemma reduce_toChargeMap_eq [source]

```
lemma reduce_toChargeMap_eq (x : TenQuanta 𝒵) :
    x.reduce.toChargeMap = x.toChargeMap
```

lemma mem_powerset_sum_of_mem_reduce_toFluxesTen [source]

```
lemma mem_powerset_sum_of_mem_reduce_toFluxesTen {F : TenQuanta 𝒵}
    {f : Fluxes} (hf : f ∈ F.reduce.toFluxesTen) :
    f ∈ (Multiset.powerset F.toFluxesTen).map fun s => s.sum
```

lemma mem_powerset_sum_of_mem_reduce_toFluxesTen_filter [source]

```
lemma mem_powerset_sum_of_mem_reduce_toFluxesTen_filter {F : TenQuanta 𝒵}
    {f : Fluxes} (hf : f ∈ F.reduce.toFluxesTen) :
    f ∈ (F.toFluxesTen.powerset.filter fun s => s ≠ 0).map fun s => s.sum
```

lemma reduce_numChiralU_of_mem_elemsNoExotics [source]

```
lemma reduce_numChiralU_of_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen.numChiralU = 3
```

lemma reduce_numAntiChiralU_of_mem_elemsNoExotics [source]

```
lemma reduce_numAntiChiralU_of_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen.numAntiChiralU = 0
```

lemma reduce_numChiralQ_of_mem_elemsNoExotics [source]

```
lemma reduce_numChiralQ_of_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen.numChiralQ = 3
```

lemma reduce_numAntiChiralQ_of_mem_elemsNoExotics [source]

```
lemma reduce_numAntiChiralQ_of_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen.numAntiChiralQ = 0
```

lemma reduce_numChiralE_of_mem_elemsNoExotics [source]

```
lemma reduce_numChiralE_of_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen.numChiralE = 3
```

lemma reduce_numAntiChiralE_of_mem_elemsNoExotics [source]

```
lemma reduce_numAntiChiralE_of_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen.numAntiChiralE = 0
```

lemma reduce_noExotics_of_mem_elemsNoExotics [source]

```
lemma reduce_noExotics_of_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen.NoExotics
```

lemma reduce_mem_elemsNoExotics [source]

```
lemma reduce_mem_elemsNoExotics {F : TenQuanta 𝒵}
    (hx : F.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    F.reduce.toFluxesTen ∈ FluxesTen.elemsNoExotics
```

def decomposeFluxes [source]

The decomposition of a relevant flux into `⟨1, 0⟩`, `⟨1, 1⟩` *and* `⟨1, -1⟩` .

```
def decomposeFluxes (f : Fluxes) : Multiset Fluxes
```

lemma decomposeFluxes_sum_of_noExotics [source]

```
lemma decomposeFluxes_sum_of_noExotics (f : Fluxes) (hf : ∃ F ∈ FluxesTen.elemsNoExotics, f ∈ F
    ) :
    (decomposeFluxes f).sum = f
```

def decompose [source]

The decomposition of a `TenQuanta` *into a* `TenQuanta` *which has the same* `reduce` *by has fluxes* `{⟨1, 0⟩, ⟨1, 0⟩, ⟨1, 0⟩}` *or* `{⟨1, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩}` *only.*

This only works for fluxes which have no exotics or zeros.

```
def decompose (x : TenQuanta 𝒵) : TenQuanta 𝒵
```

lemma decompose_add [source]

```
lemma decompose_add (x y : TenQuanta 𝒵) :
    (x + y).decompose = x.decompose + y.decompose
```

lemma decompose_filter_charge [source]

```
lemma decompose_filter_charge [DecidableEq 𝒵] (x : TenQuanta 𝒵) (q : 𝒵) :
    (x.decompose).filter (fun p => p.1 = q) =
    decompose (x.filter (fun p => p.1 = q))
```

lemma decompose_toChargeMap [source]

```
lemma decompose_toChargeMap [DecidableEq 𝒵] (x : TenQuanta 𝒵)
    (hx : x.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    x.decompose.toChargeMap = x.toChargeMap
```

lemma decompose_toCharges_dedup [source]

```
lemma decompose_toCharges_dedup [DecidableEq 𝒵] (x : TenQuanta 𝒵)
    (hx : x.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    x.decompose.toCharges.dedup = x.toCharges.dedup
```

lemma decompose_reduce [source]

```
lemma decompose_reduce (x : TenQuanta 𝒵) [DecidableEq 𝒵]
    (hx : x.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    x.decompose.reduce = x.reduce
```

lemma decompose_toFluxesTen [source]

```
lemma decompose_toFluxesTen (x : TenQuanta 𝒵)
    (hx : x.toFluxesTen ∈ FluxesTen.elemsNoExotics) :
    x.decompose.toFluxesTen = {⟨1, 0⟩, ⟨1, 0⟩, ⟨1, 0⟩} ∨
    x.decompose.toFluxesTen = {⟨1, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩}
```

def liftCharge [source]

Given a finite set of charges `c` the `TenQuanta` which do not have exotics, duplicate charges or zero fluxes, which map down to `c`. This is defined to be as efficient as possible.

```
def liftCharge (c : Finset 𝒵) : Multiset (TenQuanta 𝒵)
```

lemma toCharge_toFinset_of_mem_liftCharge [source]

```
lemma toCharge_toFinset_of_mem_liftCharge (c : Finset 𝒵)
    {x : TenQuanta 𝒵} (h : x ∈ liftCharge c) :
```

```
    x.toCharges.toFinset = c
```

lemma toCharges_nodup_of_mem_liftCharge [source]

```
lemma toCharges_nodup_of_mem_liftCharge (c : Finset ℤ) {x : TenQuanta ℤ}
    (h : x ∈ liftCharge c) : x.toCharges.Nodup
```

lemma exists_toCharges_toFluxesTen_of_mem_liftCharge [source]

```
lemma exists_toCharges_toFluxesTen_of_mem_liftCharge (c : Finset ℤ)
    {x : TenQuanta ℤ} (h : x ∈ liftCharge c) :
    ∃ a : TenQuanta ℤ, a.reduce = x ∧ a.toCharges.toFinset = c ∧
    (a.toFluxesTen = {⟨1, 0⟩, ⟨1, 0⟩, ⟨1, 0⟩}
    ∨ a.toFluxesTen = {⟨1, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩})
```

lemma mem_liftCharge_of_exists_toCharges_toFluxesTen [source]

```
lemma mem_liftCharge_of_exists_toCharges_toFluxesTen (c : Finset ℤ) {x : TenQuanta ℤ}
    (h :∃ a : TenQuanta ℤ, a.reduce = x ∧ a.toCharges.toFinset = c ∧
    (a.toFluxesTen = {⟨1, 0⟩, ⟨1, 0⟩, ⟨1, 0⟩}
    ∨ a.toFluxesTen = {⟨1, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩})) :
    x ∈ liftCharge c
```

lemma mem_liftCharge_iff_exists [source]

```
lemma mem_liftCharge_iff_exists (c : Finset ℤ) {x : TenQuanta ℤ} :
    x ∈ liftCharge c ↔
    ∃ a : TenQuanta ℤ, a.reduce = x ∧ a.toCharges.toFinset = c ∧
    (a.toFluxesTen = {⟨1, 0⟩, ⟨1, 0⟩, ⟨1, 0⟩}
    ∨ a.toFluxesTen = {⟨1, 1⟩, ⟨1, -1⟩, ⟨1, 0⟩})
```

lemma hasNoZero_of_mem_liftCharge [source]

```
lemma hasNoZero_of_mem_liftCharge (c : Finset ℤ) {x : TenQuanta ℤ}
    (h : x ∈ liftCharge c) : x.toFluxesTen.HasNoZero
```

lemma noExotics_of_mem_liftCharge [source]

```
lemma noExotics_of_mem_liftCharge (c : Finset ℤ) (F : TenQuanta ℤ)
    (h : F ∈ liftCharge c) :
    F.toFluxesTen.NoExotics
```

lemma mem_liftCharge_of_mem_noExotics_hasNoZero [source]

```
lemma mem_liftCharge_of_mem_noExotics_hasNoZero (c : Finset ℤ) {x : TenQuanta ℤ}
    (h1 : x.toFluxesTen.NoExotics) (h2 : x.toFluxesTen.HasNoZero)
    (h3 : x.toCharges.toFinset = c) (h4 : x.toCharges.Nodup) :
    x ∈ liftCharge c
```

lemma mem_liftCharge_iff [source]

```
lemma mem_liftCharge_iff (c : Finset ℤ) (x : TenQuanta ℤ) :
    x ∈ liftCharge c ↔ x.toFluxesTen ∈ FluxesTen.elemsNoExotics
      ∧ x.toCharges.toFinset = c ∧ x.toCharges.Nodup
```

lemma map_liftCharge [source]

```
lemma map_liftCharge {ℤ ℤ1 : Type}[DecidableEq ℤ] [DecidableEq ℤ1] [CommRing ℤ] [CommRing
    ℤ1]
    (f : ℤ →+* ℤ1) (c : Finset ℤ) (F : TenQuanta ℤ) (h : F ∈ liftCharge c) :
    TenQuanta.reduce (F.map fun y => (f y.1, y.2)) ∈ liftCharge (c.image f)
```

def anomalyCoefficient [source]

The anomaly coefficient of a `TenQuanta` is given by the pair of integers: `(∑ᵢ qᵢ Nᵢ, 3 * ∑ᵢ qᵢ² Nᵢ)`.

The first components is for the mixed U(1)-MSSM, see equation (22) of arXiv:1401.5084. The second component is for the mixed U(1)Y-U(1)-U(1) gauge anomaly, see equation (23) of arXiv:1401.5084.

```
def anomalyCoefficient (F : TenQuanta ℤ) : ℤ × ℤ
```

lemma anomalyCoefficient_of_map [source]

```
@[simp]
lemma anomalyCoefficient_of_map {ℤ ℤ1 : Type} [CommRing ℤ] [CommRing ℤ1]
    (f : ℤ →+* ℤ1) (F : TenQuanta ℤ) :
    TenQuanta.anomalyCoefficient (F.map fun y => (f y.1, y.2) : TenQuanta ℤ1) =
    (f.prodMap f) F.anomalyCoefficient
```

lemma anomalyCoefficient_of_reduce [source]

```
lemma anomalyCoefficient_of_reduce [DecidableEq ℤ] (F : TenQuanta ℤ) :
    F.reduce.anomalyCoefficient = F.anomalyCoefficient
```

2.53 PhysLean.Thermodynamics.Temperature.Basic

PhysLean/Thermodynamics/Temperature/Basic.lean on GitHub.

lemma ofNNReal_val [source]

Build a `Temperature` directly from a nonnegative real.

```
@[simp]
lemma ofNNReal_val (t : ℝ≥0) : (ofNNReal t).val = t
```

lemma coe_ofNNReal_coe [source]

```
@[simp]
lemma coe_ofNNReal_coe (t : ℝ≥0) : ((ofNNReal t : Temperature) : ℝ≥0) = t
```

lemma coe_ofNNReal_real [source]

```
@[simp]
lemma coe_ofNNReal_real (t : ℝ≥0) : (((t) : Temperature) : ℝ) = t
```

def ofRealNonneg [source]

Convenience: build a temperature from a real together with a proof of nonnegativity.

```
@[simp]
noncomputable def ofRealNonneg (t : ℝ) (ht : 0 ≤ t) : Temperature
```

lemma ofRealNonneg_val [source]

```
@[simp]
lemma ofRealNonneg_val {t : ℝ} (ht : 0 ≤ t) :
    (ofRealNonneg t ht).val = (t, ht)
```

def betaFromReal [source]

Map a real `t` to the inverse temperature `β` corresponding to the temperature `Real.toNNReal t` (`max t 0`), returned as a real number.

```
noncomputable def betaFromReal (t : ℝ) : ℝ
```

lemma beta_fun_T_formula [source]

Explicit closed-form for `Beta_fun_T t` when `t > 0`.

```
lemma beta_fun_T_formula (t : ℝ) (ht : 0 < t) :
    betaFromReal t = 1 / (kB * t)
```

lemma beta_fun_T_eq_on_Ioi [source]

On `Ioi 0`, `Beta_fun_T t` equals `1 / (kB * t)`.

```
lemma beta_fun_T_eq_on_Ioi :
    EqOn betaFromReal (fun t : ℝ => 1 / (kB * t)) (Set.Ioi 0)
```

lemma deriv_beta_wrt_T [source]

```
lemma deriv_beta_wrt_T (T : Temperature) (hT_pos : 0 < T.val) :
    HasDerivWithinAt betaFromReal (-1 / (kB * (T.val : ℝ)^2)) (Set.Ioi 0) (T.val : ℝ)
```

lemma chain_rule_T_beta [source]

*Chain rule for $\beta(T)$: d/dT F(β(T)) = F'(β(T)) * (-1 / (kB * T^2)), within `Ioi 0`.*

```
lemma chain_rule_T_beta {F : ℝ → ℝ} {F' : ℝ}
    (T : Temperature) (hT_pos : 0 < T.val)
    (hF_deriv : HasDerivWithinAt F F' (Set.Ioi 0) (T.β : ℝ)) :
    HasDerivWithinAt (fun t : ℝ => F (betaFromReal t))
      (F' * (-1 / (kB * (T.val : ℝ)^2))) (Set.Ioi 0) (T.val : ℝ)
```

2.54 PhysLean.Units.Basic

PhysLean/Units/Basic.lean on GitHub.

class HasDim [source]

This typeclass indicates that there is a dimension `dim M : Dimension` associated with the type `M`.

```
class HasDim (M : Type) where
  d : Dimension
```

class abbrev [source]

A type `M` carries a dimension `d` if every element of `M` is supposed to have this dimension. For example, the type `Time` will carry a dimension `T`.

```
class abbrev CarriesDimension (M : Type) := HasDim M, MulAction ℝ≥0 M
```

2.55 PhysLean.Units.Integral

PhysLean/Units/Integral.lean on GitHub.

instance MulUnitDependent (MeasureTheory.Measure M) [source]

```
noncomputable instance (M : Type)
    [NormedAddCommGroup M] [NormedSpace ℝ M] [HasDim M]
    [MeasurableSpace M] [self : MeasurableConstSMul ℝ M] :
      MulUnitDependent (MeasureTheory.Measure M) where
```

2.56 PhysLean.Units.UnitDependent

PhysLean/Units/UnitDependent.lean on GitHub.

lemma scaleUnit_apply [source]

```
lemma HasDim.scaleUnit_apply {M : Type} [CarriesDimension M]
    (u1 u2 : UnitChoices) (m : M) :
    scaleUnit u1 u2 m = (u1.dimScale u2 (dim M)) • m
```

instance LinearUnitDependent M [source]

```
noncomputable instance {M : Type} [AddCommMonoid M] [Module ℝ M] [HasDim M] :
    LinearUnitDependent M where
```

instance ContinuousLinearUnitDependent M [source]

```
noncomputable instance {M : Type} [AddCommMonoid M] [Module ℝ M]
    [HasDim M] [TopologicalSpace M]
    [ContinuousConstSMul ℝ M] : ContinuousLinearUnitDependent M where
```

2.57 PhysLean.Units.WithDim.Basic

PhysLean/Units/WithDim/Basic.lean on GitHub.

instance HasDim (WithDim d M) [source]

```
instance (d : Dimension) (M : Type) : HasDim (WithDim d M) where
```

lemma dim_apply [source]

```
@[simp]
lemma dim_apply (d : Dimension) (M : Type) :
    dim (WithDim d M) = d
```

2.58 scripts.MetaPrograms.redundant_imports

scripts/MetaPrograms/redundant_imports.lean on GitHub.

def RedundantImports [source]

```
def Imports.RedundantImports (imp : Import) : MetaM UInt32
```

2.59 scripts.MetaPrograms.runPhysLeanLinters

scripts/MetaPrograms/runPhysLeanLinters.lean on GitHub.

def runLinterOnModule [source]

Run the Batteries linter on a given module and update the linter if `update` is `true`.

```
unsafe def runLinterOnModule  (module : Name): IO Unit
```

2.60 scripts.MetaPrograms.spelling

scripts/MetaPrograms/spelling.lean on GitHub.

def moduleDocs [source]

The strings appearing as documentation within PhysLean.

```
def moduleDocs : MetaM (Array String)
```

def allWords [source]

All the words in either module doc-strings or doc-strings of definitions, theorems and lemmas.

```
def allWords : MetaM (Array String)
```

def dictionary [source]

The custom dictionary of correctly spelled words.

```
def dictionary : MetaM (Array String)
```