Developing Physics Extensions for MeanField

A worked ideal-gas plus radiation equation of state

Physics developer example module

Code checkpoint `71423d543f1775d3b943d9c0361f49edd56b59c1`
Manual built 2026-09-06

What this manual promises. You will implement a real two-variable equation of state, make its physical relations and Jacobian derivatives compile-time contracts, and compose it with the existing isobaric surface, fixed-total-mass invariant, and fixed-angular-momentum invariant. You do not need to write block indices or repeat the model assembly logic.

What this manual does not claim. The current numerical stellar-equilibrium core is barotropic. The worked EOS depends independently on density and temperature. It therefore forms a valid typed stellar-model specification, but it cannot yet be passed to the current `discretize` operation. A temperature or entropy field and a governing thermal equation must be added to the equilibrium formulation first. The example encodes this boundary as a compile-time assertion.

Contents

1 Audience and objective

This manual is for an astronomer or physicist who knows the equations they want to add but may not be interested in the template machinery used to assemble a nonlinear root system. The central design rule is:

> State the physics once, in physics vocabulary. Let the library infer the structural type of the model, residual, Jacobian, normalization plan, and preconditioner wherever a runtime implementation exists. If a required piece does not exist, reject the operation at compile time.

The worked extension lives entirely under `extension_example/`. It does not modify a source file in `libmeanfield/`. This separation matters. It demonstrates the public extension surface rather than relying on privileged access to the implementation.

At the end, you should be able to answer four questions for a proposed physics extension:

1. What quantities and relations does the new physics provide?
2. Which analytic derivatives does a Newton Jacobian require?
3. Which existing specifications can be composed with it immediately?
4. Is the complete numerical runtime available, or is only the symbolic model declaration available?

2 The mental model: five layers

It helps to separate five ideas that are often blended together in scientific software.

Layer	**Physics question**	**MeanField representation**
Quantity	What does this scalar mean?	A type such as `Density`, `Temperature`, or `Pressure`, wrapped in a `QuantityValue`.
Relation	Which variables determine which result?	A compile-time sentence such as $P = P(\rho, T)$, represented by `eos::Relation<Pressure, Density, Temperature>`.
Specification	What physical choice did the model make?	A concrete EOS, surface condition, invariant, phase condition, gauge, or rotation law with a small `ModelDefinition`.
Compiled system	Which unknowns, residuals, and Jacobian couplings follow from the selected specifications?	A type inferred from the complete, variadic specification pack.
Runtime provider	How are those residuals and derivatives evaluated on a mesh?	Numerical assembly, normalization, and preconditioning code that must exist for the compiled type.

A model may be valid at one layer and intentionally unavailable at a later layer. In this example, $P(\rho, T)$ is a valid EOS relation and the four chosen specifications form a valid stellar-model type. The present runtime provider has a barotropic material core, so it rejects the thermal model before a discretization object can be formed.

This distinction prevents two dangerous outcomes:

- silently dropping a supplied physical constraint; and
- pretending a multidimensional EOS is barotropic by choosing an undocumented path through thermodynamic state space.

3 Map of the example module

File	Purpose
`ideal_gas_radiation.cppm`	Declares relations, implements the EOS, analytic derivatives, input validation, and compile-time protocol checks.
`rotating_stellar_model.cppm`	Provides the small physics-facing composition wrapper and documents the current runtime boundary.
`demo.cpp`	Evaluates one CGS thermodynamic state and constructs the typed stellar-model specification.
`tests/ideal_gas_radiation.cpp`	Tests dimensions, thermodynamic identities, scaling laws, derivative accuracy, and domain handling.
`tests/rotating_stellar_model.cpp`	Tests inferred roles, exact specification types, values, and capability rejection.
`manual/physics_developer_manual.tex`	The source of this manual.

The module has its own library, demonstration executable, and test executable. It links to MeanField, but its files are not added to the `mean_field` library target and its tests are not added to the main regression executable.

4 Worked physics: ideal gas plus radiation

4.1 Assumptions

The example describes a deliberately simple local thermodynamic model:

- matter is a classical, nondegenerate, monatomic ideal gas;
- the mean molecular weight μ is fixed;
- gas and radiation share a single temperature T;
- radiation is in local thermodynamic equilibrium;
- ionization, composition evolution, pairs, degeneracy, Coulomb corrections, and opacity do not enter the EOS.

These assumptions are appropriate for demonstrating the extension protocol. They are not a universal stellar EOS.

Let k_{B} be Boltzmann's constant, m_{u} the atomic mass unit, a the radiation energy-density constant, and

$$\mathcal{R} = \frac{k_{\mathrm{B}}}{\mu m_{\mathrm{u}}} \tag{1}$$

the gas constant per unit mass. The pressure contributions are

$$P_{\mathrm{gas}} = \rho \mathcal{R} T, \tag{2}$$

$$P_{\mathrm{rad}} = \frac{a T^4}{3}, \tag{3}$$

$$P(\rho, T) = P_{\mathrm{gas}} + P_{\mathrm{rad}}. \tag{4}$$

For a monatomic gas, the specific internal energies are

$$u_{\text{gas}} = \frac{3}{2}\mathcal{R}T, \tag{5}$$

$$u_{\text{rad}} = \frac{aT^4}{\rho}, \tag{6}$$

$$u(\rho, T) = u_{\text{gas}} + u_{\text{rad}}. \tag{7}$$

Using $h = u + P/\rho$, the specific enthalpy is

$$h(\rho, T) = \frac{5}{2}\mathcal{R}T + \frac{4aT^4}{3\rho}. \tag{8}$$

4.2 Analytic derivatives

A Newton method needs derivatives of the residual with respect to its state. For this EOS, the useful local derivatives are

$$\left.\frac{\partial P}{\partial \rho}\right|_T = \mathcal{R}T, \qquad \left.\frac{\partial P}{\partial T}\right|_\rho = \rho\mathcal{R} + \frac{4aT^3}{3}, \tag{9}$$

$$\left.\frac{\partial u}{\partial \rho}\right|_T = -\frac{aT^4}{\rho^2}, \qquad \left.\frac{\partial u}{\partial T}\right|_\rho = \frac{3}{2}\mathcal{R} + \frac{4aT^3}{\rho}, \tag{10}$$

$$\left.\frac{\partial h}{\partial \rho}\right|_T = -\frac{4aT^4}{3\rho^2}, \qquad \left.\frac{\partial h}{\partial T}\right|_\rho = \frac{5}{2}\mathcal{R} + \frac{16aT^3}{3\rho}. \tag{11}$$

The subscript is not optional notation. It tells the developer what is held fixed and corresponds directly to the `WithRespectTo<...>` tag in the code. A derivative with the right numerical return type but the wrong held variable is a physics bug.

4.3 Domain and units

The implemented material domain is

$$\rho > 0, \qquad T \geq 0. \tag{12}$$

Positive density is required because the specific radiation energy and enthalpy contain $1/\rho$. The radiation constant may be set to zero for a pure-gas verification problem. Other constants and μ must be finite and positive.

MeanField's `QuantityValue` gives semantic type safety. It does not perform dimensional algebra or unit conversion. The defaults in this example are CGS:

Constant	Default	CGS unit
k_{B}	1.380649×10^{-16}	erg K^{-1}
m_{u}	$1.66053906660 \times 10^{-24}$	g
a	7.5657×10^{-15}	erg cm^{-3} K^{-4}

If the rest of a model uses nondimensional or code units, all three constants, density, temperature, and returned thermodynamic values must be transformed coherently. A typed value prevents passing temperature where density belongs; it cannot detect a CGS value mixed with a nondimensional one.

5 Implementing the EOS

5.1 Step 1: name the relations

The first code says what the EOS knows, without saying how it computes it.

```
namespace q = mean_field::dimensions::quantity;

using PressureFromDensityAndTemperature = mean_field::eos::Relation<
    q::Pressure,
    q::Density,
    q::Temperature>;

using SpecificEnthalpyFromDensityAndTemperature = mean_field::eos::Relation<
    q::SpecificEnthalpy,
    q::Density,
    q::Temperature>;
```

The first template argument is the output. Remaining arguments are inputs in call order. Consequently,

```
eos::evaluate<q::Pressure>(eosModel, density, temperature);
```

is valid, while the same call with `temperature, density` is rejected at compile time.

Use an existing quantity type when it has the intended physical meaning. Temperature and specific internal energy already exist in `mean_field::dimensions::quantity`. If a genuinely new quantity is needed, it can derive from `ThermodynamicQuantity` and provide a stable identifier, but adding a quantity to numerical normalization also requires an appropriate scale law and runtime support.

5.2 Step 2: identify the specification role

Inside the EOS class, one alias connects the physics type to the stellar-model front end:

```
using ModelDefinition = mean_field::eos::ConstitutiveLaw<
    IdealGasRadiation,
    "IdealGasRadiation">;
```

The stable name is used in compile-time keys and runtime descriptions. Choose a specific, durable name. Do not encode parameter values in it, and do not reuse the same role-name pair for unrelated physics.

This alias is the wrapper intended for physics developers. It projects to the more general model-definition machinery, but the EOS author does not need to name generated blocks, residual rows, or a normalization topology because a constitutive law owns no scalar solver border by itself.

5.3 Step 3: publish the complete relation catalog

```
using Relations = mean_field::eos::RelationCatalog<
    PressureFromDensityAndTemperature,
    SpecificInternalEnergyFromDensityAndTemperature,
    SpecificEnthalpyFromDensityAndTemperature>;
```

Treat this catalog as an API promise. Every listed relation must have an exact `evaluate` overload. A relation that exists as a helper function but is missing from the catalog is not visible to generic consumers. A relation in the catalog without a matching evaluator makes `EquationOfStateModel` false.

5.4 Step 4: implement evaluations in physics notation

The pressure overload is small because named component functions expose useful diagnostics:

```
[[nodiscard]] dimensions::PressureValue evaluate(
    PressureFromDensityAndTemperature,
    dimensions::DensityValue density,
    dimensions::TemperatureValue temperature
) const {
    return pressureContributions(density, temperature).total();
}
```

The relation tag is an empty compile-time object. It selects the overload but does not cost storage. Return the exact typed value requested by the relation. Returning a raw `double`, even with the correct number, fails the EOS concept.

The generic user-facing call is

```
const auto pressure = mean_field::eos::evaluate<
    mean_field::dimensions::quantity::Pressure>(
        equationOfState,
        density,
        temperature
    );
```

This call infers the relation from output type and typed input sequence. Client code does not construct relation tags directly.

5.5 Step 5: implement Jacobian derivatives

Each derivative overload repeats the relation and states the differentiation variable explicitly:

```
[[nodiscard]] eos::PartialDerivative<q::Pressure, q::Temperature>
partialDerivative(
    PressureFromDensityAndTemperature,
    eos::WithRespectTo<q::Temperature>,
    dimensions::DensityValue density,
    dimensions::TemperatureValue temperature
) const {
    const double T = temperature.value();
    return eos::PartialDerivative<q::Pressure, q::Temperature>{
        density.value() * specificGasConstant() +
        4.0 * parameters().radiationConstant * T * T * T / 3.0
    };
}
```

The important contracts are:

- relation inputs occur in the same order as in `evaluate`;
- `WithRespectTo<Temperature>` identifies the independent variable;
- the return type is exactly `PartialDerivative<Pressure, Temperature>`; and
- the formula is a partial derivative at fixed density.

5.6 Step 6: make the compiler check the public claim

The source finishes with assertions such as

```
static_assert(eos::EquationOfStateModel<IdealGasRadiation>);

static_assert(eos::SupportsPartialDerivative<
    IdealGasRadiation,
    PressureFromDensityAndTemperature,
    q::Temperature>);
```

These assertions are not substitutes for numerical tests. They prove shape: the relation is declared, the argument order matches, and the exact output type exists. Numerical tests must still prove that the derivative computes the right mathematics.

6 Composing a rotating stellar model

The model itself is the direct expression of the physical choices:

```
auto model = mean_field::model::StellarModel(
    IdealGasRadiation(eosParameters),
    mean_field::surface::Isobaric({.Psurf = surfacePressure}),
    mean_field::integral::FixedTotalMass({.Mtotal = totalMass}),
    mean_field::integral::FixedAngularMomentum({
        .Jtotal = totalAngularMomentum,
        .axis = rotationAxis,
        .center = rotationCenter
    })
);
```

The example wraps this expression in `makeRotatingStellarModel`. The wrapper groups frequently used inputs; it does not duplicate model assembly. An advanced caller can always use `StellarModel(...)` directly.

The compiler infers four distinct specification types and their roles:

Specification	Role	Structural contribution
`IdealGasRadiation`	constitutive law	EOS relations; no generated scalar coordinate
`Isobaric`	boundary condition	fixed surface pressure; no generated scalar coordinate
`FixedTotalMass`	invariant	mass constraint residual and its scalar multiplier
`FixedAngularMomentum`	invariant	angular-momentum constraint residual and angular velocity as a physical coordinate

The two invariants each contribute one scalar unknown and one scalar residual. The complete symbolic operator remains square. Reordering the constructor arguments does not define a new physical combination: the library canonicalizes the specification set by stable compile-time keys while storing exactly one object of each inferred type.

The model exposes physics-facing accessors:

```
model.equationOfState();
model.surfaceCondition();
model.specification<integral::FixedTotalMass>();
model.specification<integral::FixedAngularMomentum>();
```

These preserve exact types. There is no base-class downcast and no string lookup in the inner numerical path.

7 What compiles today, and why

7.1 The three useful questions

For a new model, ask these separately:

1. **Is each object a valid specification?** The type has a valid role definition, parameters, and contribution metadata.
2. **Can the objects form a stellar-model type?** Their stable keys are unique, exactly one constitutive law is present, and generated scalar arities make a square symbolic system.
3. **Can the current numerical backend discretize that exact type?** Every core equation, surface conversion, residual assembly, Jacobian coupling, normalization action, and runtime provider exists.

   For this example, the answers are yes, yes, and no.

7.2 Why the current core rejects this EOS

The current stellar material core is barotropic. Its state can close density through a relation of the form

$$\rho = \rho(h), \tag{13}$$

with the corresponding derivative $\mathrm{d}\rho/\mathrm{d}h$. This is sufficient for a polytropic barotrope.
The example instead supplies

$$P = P(\rho, T), \qquad u = u(\rho, T), \qquad h = h(\rho, T). \tag{14}$$

Hydrostatic balance plus an isobaric surface does not determine both ρ and T throughout the star. A second physical equation is required. Depending on the intended model, it might be an entropy prescription, radiative energy transport, convective closure, or a full energy equation with sources and sinks.

The example therefore includes

```
static_assert(model::StellarModelType<RotatingStellarModel>);
static_assert(!equilibrium::StellarEquilibriumModel<RotatingStellarModel>);
```

The second assertion is a capability tripwire. When a thermal equilibrium core is eventually implemented, the assertion and this manual must be revised together.

7.3 What must be added for a coupled thermal solve

A physically complete extension will need all of the following, designed as a single compile-time path rather than a special case for this EOS:

1. A selected thermal state coordinate, such as temperature or specific entropy, with a finite-element field family.
2. A thermal residual equation whose physical closure is explicit.
3. EOS relations and partial derivatives required by both hydrostatic and thermal residuals.

4. Surface data appropriate to the thermal equation, such as a fixed temperature, luminosity, or atmosphere matching condition.
5. Generated block and Jacobian couplings inferred from the selected thermodynamic equation set.
6. Normalization metadata for the new field and residual, including a physically meaningful Riesz map and scale.
7. A preconditioner component that approximates the new thermal block and its important coupling to structure.
8. Runtime providers for assembly and linearization of every inferred contribution.

The EOS in this module is already expressed in the multi-input relation vocabulary needed by such a future core. It should not be rewritten as a bespoke EOS-backend combination.

8 Testing a physics extension

A robust extension test suite has four layers. Passing only the first layer is not enough.

8.1 Compile-time protocol tests

These answer structural questions:

- Is the class a self-describing constitutive-law specification?
- Does every catalog relation have an exact evaluator?
- Are required partial derivatives available?
- Are input order and output quantity types enforced?
- Does composition preserve the exact EOS, surface, and invariant types?
- Is the inferred scalar root system square?

The reversed pressure call (T, ρ) is tested as a non-callable expression. This is a useful negative contract: it proves that strong quantity types catch an error that two raw `double` arguments would not catch.

8.2 Exact worked state

The tests choose artificial constants that make hand calculation easy:

$$\mu = 2, \quad k_{\mathrm{B}} = 12, \quad m_{\mathrm{u}} = 3, \quad a = 9, \tag{15}$$

so $\mathcal{R} = 2$. At $\rho = 4$ and $T = 2$,

$$P_{\mathrm{gas}} = 16, \qquad P_{\mathrm{rad}} = 48, \qquad P = 64, \tag{16}$$

$$u_{\mathrm{gas}} = 6, \qquad u_{\mathrm{rad}} = 36, \qquad u = 42, \tag{17}$$

$$h = u + P/\rho = 42 + 16 = 58. \tag{18}$$

This single state catches wrong factors of 1/3, 3/2, 4/3, and 5/2. It also tests the independent thermodynamic identity $h = u + P/\rho$.

8.3 Scaling and limiting behavior

The tests verify transformations more diagnostic than a list of isolated numbers:

- doubling ρ doubles P_{gas};
- changing ρ leaves P_{rad} unchanged;
- doubling T doubles P_{gas};
- doubling T multiplies P_{rad} by 16; and
- at $T_{\text{cross}} = (3\rho\mathcal{R}/a)^{1/3}$, gas and radiation pressures agree.

These claims exercise the physical exponents and parameter dependencies. They would fail even if a few reference values happened to be fitted correctly.

8.4 Analytic versus numerical derivatives

Every one of the six analytic partial derivatives is compared with a centered difference at three materially different states. For a scalar function f,

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}. \tag{19}$$

The test uses

$$\epsilon = \epsilon_{\text{machine}}^{1/3} \max(1, |x|). \tag{20}$$

For a centered difference, this scale balances truncation and roundoff for smooth double-precision functions. The comparison tolerance is selected from the observed error order and is much tighter than a solver tolerance. It tests the local Jacobian implementation, not convergence of a nonlinear solve.

When extending a tabulated or iterative EOS, use tolerances justified by that algorithm's interpolation and inner-solve error. Do not copy the analytic-EOS tolerance mechanically.

8.5 Domain tests

Tests require rejection of invalid constants, nonfinite values, zero density, and negative temperature. Domain tests should check error categories where a caller can recover differently from a nonfinite input and an out-of-domain state.

9 Adding other kinds of physics

The same pattern extends beyond an EOS: declare a physics-facing wrapper, state dependencies and effects once, and let the compiler generate structure. The details below are a map, not a replacement for tests of the actual equations.

9.1 Surface conditions

A surface condition uses

```
using ModelDefinition = mean_field::surface::BoundaryCondition<
    MySurfaceCondition,
    "MySurfaceCondition">;
```

It should name its physical target quantity and typed target value. A new surface condition can enter a `StellarModel` as a distinct type before a runtime surface compiler exists. Full discretization is available only if the chosen EOS can convert the boundary statement into the thermodynamic variable used by the selected material formulation.

For example, an isothermal surface is physically meaningful for a thermal model. It does not become meaningful for a barotropic formulation merely because both types compile as specifications.

9.2 Integral invariants

An invariant such as fixed mass or fixed angular momentum owns a global scalar equation and usually a conjugate scalar coordinate. Physics-facing aliases include `FixedScalarWithMultiplier` and `FixedScalarWithPhysicalCoordinate`. Their declarations identify:

- the target physical quantity;
- the generated coordinate quantity;
- the constraint residual quantity;
- which state quantities the integral reads;
- which equations the generated coordinate changes;
- stable symbols and identifiers; and
- normalization metadata.

That information is enough to generate the border shape and required Jacobian incidence. A runtime provider must still implement the integral, its derivative, and its action on affected equations. If that provider is absent, compilation must stop at the operation boundary rather than ignore the invariant.

9.3 Phase conditions

A phase condition removes a neutral direction or selects one representative of a family. It is not necessarily a conserved physical quantity. Fixed central density, for example, can serve as a phase constraint while fixed total mass is the physically informed invariant.

Use `ScalarPhaseCondition` for the physics-facing declaration. Keep the distinction explicit in naming, documentation, and tests: an invariant changes the physical problem, while a phase condition selects a solution within a degenerate representation.

10 Normalization and preconditioning

Normalization and preconditioning solve different problems.

The normalization operator gives each field and residual a physically meaningful inner product and scale. Conceptually, it maps the raw root system to coordinates where comparisons such as residual norms and line-search acceptance are not dominated merely by units. A pressure-like row near 10^{16} and a dimensionless row near unity should not be compared as raw numbers.

The preconditioner approximates the inverse action of the scaled Jacobian. It is concerned with coupling and spectral difficulty, not just magnitude. A good normalization can improve the numerical setting in which the preconditioner operates, but it cannot replace an approximation to elliptic, material, surface, or global-border couplings.

For a new generated scalar, the specification must provide normalization metadata that is dimensionally coherent with its target, coordinate, and residual quantities. For a new distributed thermal field, the discretization must select a compatible topology and physical scale. If no scale law exists, the correct

outcome is a compile-time rejection of normalized discretization, not an identity scale inserted without documentation.

Physics tests for a new normalization should include:

- invariance under the intended change of physical units;
- expected mesh-refinement behavior of the discrete norm;
- exact results for constant or low-order fields when available;
- consistent scaling of a residual and its Jacobian action; and
- quantified interaction with the chosen preconditioner.

11 A practical development workflow

1. **Write the equations first.** List state variables, parameters, outputs, domains, held-fixed variables, and expected limits.
2. **Reuse or define quantity types.** Avoid raw scalars at public physics boundaries.
3. **Declare the smallest honest relation catalog.** Do not promise an inverse relation that is multivalued or requires an undocumented closure.
4. **Implement evaluations and analytic derivatives together.** Keeping them adjacent makes sign and factor audits easier.
5. **Add compile-time assertions.** Check the EOS concept, each required derivative, the exact composed model type, and negative capability cases.
6. **Add hand-calculated physics tests.** Use states that expose every coefficient and term.
7. **Add property tests.** Check scaling laws, limits, monotonicity, conservation identities, symmetry, or convexity as appropriate.
8. **Compare derivatives numerically.** Cover the relevant state domain, not just one comfortable point.
9. **Compose with several existing specifications.** Confirm that the model infers roles and preserves exact types without a bespoke combination.
10. **Ask the runtime capability question explicitly.** If false, document which physical equation or provider is absent.
11. **Build only the extension target while iterating.** Run the wider regression suite only when shared library code changes.

12 Build and run

From the repository root, use the configured build directory and request only the extension targets:

```
cmake --build cmake-build-profile-homebrew-llvm \
  --target extension_example_demo extension_example_tests
```

Run only the extension tests:

```
./cmake-build-profile-homebrew-llvm/extension_example/extension_example_tests
```

Run the demonstration:

```
./cmake-build-profile-homebrew-llvm/extension_example/extension_example_demo
```

Compile this manual into the build directory:

```
cmake --build cmake-build-profile-homebrew-llvm \
  --target extension_example_manual
```

The extension tests use a separate Catch2 executable. Running it does not run the main MeanField test suite.

13 Gotchas

1. **A typed scalar is not a unit library.** A `TemperatureValue` says "temperature," not "kelvin." Keep the whole parameter set in one coherent unit system.
2. **Relation input order is part of the type.** $P(\rho, T)$ and $P(T, \rho)$ are different relation types even if a human could reorder the arguments.
3. **The catalog is authoritative.** An overload omitted from `Relations` is invisible to generic dispatch. A catalog entry without an exact overload invalidates the EOS concept.
4. **State what is held fixed.** A total derivative along an isentrope is not the same as a partial derivative at fixed temperature.
5. **Do not invent an inverse closure.** A two-variable EOS does not generally supply $\rho(h)$ without another thermodynamic condition.
6. **Specification success is not runtime success.** A valid `StellarModel` type can be rejected by `discretize` when a required numerical provider is absent.
7. **A surface statement must match the formulation.** Fixed pressure alone does not set both density and temperature at a thermal surface.
8. **Stable names are identities.** Keep them unique within a role and stable across harmless refactors.
9. **An invariant is not a phase condition.** Fixed total mass is physically informative; fixed central density may be used to select a phase. Their generated coordinates have different meanings.
10. **Analytic formulas still need numerical audits.** Templates can prove that $\partial P/\partial T$ exists, not that a factor of four is correct.
11. **Avoid EOS-specific solver branches.** Extend generic relation, equation, normalization, and provider mechanisms so future EOS types follow the same path.
12. **Validate before powers or division.** Nonfinite inputs and invalid density should fail with an EOS-domain error rather than propagate a NaN through a global residual.

14 Review checklists

14.1 EOS author checklist

☐ Assumptions and validity domain are written down.

☐ Parameters have physical names and coherent units.

☐ Public inputs and outputs use quantity types.

☐ The relation catalog is minimal and complete.

☐ Every catalog relation has an exact evaluator.

☐ Every required Jacobian partial has the correct held-fixed variable and return type.

☐ Nonfinite and out-of-domain states are rejected.

☐ Hand-worked values test all coefficients.

☐ Physical scaling laws and limiting regimes are tested.

☐ Analytic derivatives agree with numerical derivatives.

14.2 Coupled-model checklist

☐ Exactly one constitutive law is present.

☐ Surface and EOS relations are thermodynamically compatible.

☐ Every supplied invariant and phase condition appears in the inferred model type.

☐ Generated unknown and residual arities are square.

☐ All Jacobian incidences required by declared dependencies and effects are present.

☐ Every new coordinate and residual has normalization metadata and a runtime action.

☐ The preconditioner includes the important new couplings.

☐ Runtime capability is asserted before discretization.

☐ Failure of an unsupported combination occurs at compile time with a useful boundary and message.

15 Glossary

Term	Meaning in this code base
Barotropic EOS	An EOS whose thermodynamic closure can be expressed with one independent material coordinate for the current problem, such as $\rho = \rho(h)$.
Boundary condition	A physical statement applied at a domain boundary, such as fixed surface pressure. It is a model specification role.
Canonical specification set	The order-independent compile-time collection of the exact specification types supplied to `StellarModel`.
Compile-time contract	A property checked by C++ type formation or a concept, before a numerical run begins.
Constitutive law	A specification that relates material state quantities. An EOS is the principal example.
Discretization	The mesh, finite-element spaces, ordering, normalization prescription, and runtime data needed to turn a compiled physical system into a finite-dimensional root problem.
Equation of state (EOS)	A set of thermodynamic relations with a declared validity domain and parameters.
Generated coordinate	A scalar unknown introduced by a specification, such as the mass-constraint multiplier or angular velocity associated with fixed angular momentum.
Invariant	A physically prescribed global quantity enforced by a residual, such as total mass or total angular momentum.
Jacobian	The derivative of the full residual vector with respect to the full state vector. Its block structure is inferred from compiled equations and specification contributions.
Model specification	One exact physical choice carrying a role, stable key, parameters, and structural contribution metadata.
Normalization	A physically informed mapping that supplies comparable coordinates and residual norms across fields with different units, topologies, and magnitudes.
Partial derivative	A derivative with respect to one declared relation input while its other inputs are held fixed.
Phase condition	A condition that removes a neutral direction or chooses one representative from a family of equivalent solutions. It is distinct from a physical invariant.
Physical Riesz map	The discrete map induced by a selected physical inner product and scale. It connects a field or residual with its dual coordinate in a topology-aware way.
Preconditioner	An efficient approximation to the inverse scaled Jacobian, used to make the linearized solve tractable.

Term	Meaning in this code base
Quantity type	A zero-storage type that names physical meaning, such as `Temperature`. A `QuantityValue<Temperature>` stores the scalar.
Relation catalog	The complete compile-time list of input-output relations an EOS claims to implement.
Residual	One equation written as a quantity that should be zero at a solution. The full nonlinear problem is a vector of residual blocks.
Runtime provider	Numerical code that evaluates or assembles an inferred physics contribution for a particular formulation.
Solver border	The small set of generated global scalar rows and columns coupled to the distributed physical core.
Specification role	The category of a physical choice: constitutive law, boundary condition, invariant, phase condition, gauge choice, or rotation law.
Stable name	A compile-time string used with a role to identify one specification mechanism consistently across inferred structures and runtime descriptions.
Stellar model	The strongly typed, canonical composition of the exact physical specification objects selected by a user.
Symbolically square	The compiled state and residual descriptions have equal total scalar arity before mesh-dependent sizes are known.
Thermal closure	The additional equation or prescription needed to determine an independent thermal variable such as temperature or entropy.
Typed value	A scalar wrapper whose C++ type carries its physical meaning and prevents accidental interchange of unrelated quantities.

16 Final perspective

The extension mechanism is successful when a physics developer can read the new source primarily as equations, parameters, domains, and derivatives. The compiler should then answer structural questions: whether the relations are complete, whether the selected model is square, which exact constraints are present, and whether the current numerical backend implements every inferred piece.

The ideal-gas plus radiation example intentionally reaches that boundary. It demonstrates a valid nonbarotropic EOS and a valid rotating stellar-model specification without claiming a nonexistent thermal equilibrium solve. That is the behavior future extensions should preserve: composable when supported, precisely rejected when incomplete, and always honest about the physics.