# `Tensorium_lang`: an MLIR-based DSL for tensorial systems in numerical relativity

Louis Touzalin

**Abstract**

`Tensorium_lang` is an experimental domain-specific language for making tensorial structure compiler-visible in numerical-relativity kernels. Instead of treating geometric quantities as conventions over raw arrays, the language exposes tensor variance, index structure, spacetime metrics, $3 + 1$ decompositions, and evolution surfaces to a staged compiler pipeline. The current implementation lowers this structure through a Tensorium IR and an MLIR dialect toward LLVM-compatible kernels with an explicit host ABI. The reference example is an analytic Schwarzschild BSSN constraint fixture. The system is not yet a complete numerical-relativity code or a full symbolic geometry engine; it is a compiler experiment for preserving and validating geometric information across parsing, typing, lowering, code generation, and runtime execution.

# Contents

# 1 Introduction

Numerical relativity codes manipulate objects with many layers of mathematical structure: tensor variance, index contractions, spacetime metrics, $3+1$ decompositions, gauge variables, constraints, and differential operators. In conventional C, C++, or Fortran kernels, most of that structure is lost before the compiler sees it. A field such as $\gamma_{ij}$ is typically an array with a storage convention; whether its indices are covariant, whether an expression is a valid contraction, or whether a buffer is a metric component is known to the programmer and to tests, but not to the compiler.

The main idea of `Tensorium_lang` is to make tensorial structure visible to the compiler instead of leaving it as an array-indexing convention. A Tensorium program declares geometric objects, writes indexed equations, and asks the compiler to preserve that meaning through a typed IR, an MLIR dialect, lowering passes, and a generated kernel ABI. The goal is to generate code while keeping enough information for the compiler to check tensor ranks, index roles, metric declarations, $3+1$ bindings, and kernel interfaces before numerical execution.

This paper presents the current implementation as a technical compiler prototype. It does not claim to replace established numerical-relativity frameworks, and it does not yet derive arbitrary curvature tensors or complete formulations automatically. It shows a compiler path where geometric information is kept from source notation down to executable kernels, with Schwarzschild and analytic BSSN fixtures used as regression tests.

### Contributions

This draft focuses on the following contributions:

- a source language in which tensor variance, indexed expressions, metrics, $3+1$ decompositions, and evolution surfaces are explicit;
- a typed compiler pipeline that preserves those geometric objects from parsing through Tensorium IR, Tensorium MLIR, and LLVM-compatible kernel generation;
- an Einstein-summation lowering path in which contractions become `tensorium.einsum` operations annotated with free, contracted, dangling, or invalid index roles;
- an MLIR dialect and generated ABI that keep field roles, tensor component counts, coordinate metadata, and kernel buffer access visible to the compiler and host runtime;
- regression tests showing that Schwarzschild metric data and an analytic BSSN constraint surface can be lowered and executed while preserving the intended tensor structure;
- a metric-derived geometry path that constructs Christoffel symbols and spatial Ricci tensors from $\gamma_{ij}$ and $\gamma^{ij}$;
- an explicit statement of current limits: the BSSN fixture validates an encoded analytic surface, not automatic derivation of the full BSSN formalism.

# 2 Positioning

`Tensorium_lang` sits between symbolic tensor code generation, numerical-relativity infrastructure, PDE DSLs, and compiler IR work. Its goal is narrower than a full simulation framework and more domain-specific than a general PDE language.

Systems such as Kranc and NRPy+ have shown that symbolic manipulation and code generation are useful for numerical relativity [7, 8]. Tensorium targets a different compiler boundary: tensor variance, index roles, metric declarations, $3+1$ decompositions, and generated kernel ABI

metadata remain explicit after source parsing and into the lowering pipeline. The difference is not broader symbolic coverage. It is keeping geometric meaning visible across compiler IRs and generated kernel interfaces.

Frameworks such as the Einstein Toolkit and Cactus provide established infrastructure for large-scale simulations, scheduling, parallel execution, and community formulations [9, 10]. Tensorium works below that level: it describes and lowers geometric kernels that could be embedded in such runtimes or in a future AMR backend. It is meant to sit under simulation infrastructure, not replace it.

SymPy-based workflows provide flexible symbolic algebra and code generation [11], while Devito, Firedrake, and FEniCS demonstrate the value of high-level DSLs for PDE discretization and finite-element or stencil code generation [12, 14, 13]. Tensorium adds a numerical-relativity object model to the compiler: spacetime metrics, index variance, contractions, and 3+1 bindings are first-class compiler objects rather than comments on array expressions.

### 2.1 Why MLIR

MLIR, the Multi-Level Intermediate Representation project in the LLVM ecosystem [16, 15], is a compiler infrastructure for building staged lowering pipelines. Its core idea is that a compiler does not need to jump directly from a source AST to a low-level IR. Instead, it can introduce domain-specific dialects, each with its own operations, types, attributes, verifiers, and rewrite passes, and then progressively lower those dialects toward standard loop, memory, arithmetic, and LLVM-compatible representations.

This model is useful because many scientific DSLs need an intermediate level that is neither source syntax nor machine-oriented code. Tensorium needs to talk about objects such as `metric4`, `decompose3p1_from_metric`, `tensorium.einsum`, and `!tensorium.field<f64, up, down>`. These objects are too geometric to be represented directly as ordinary scalar loops, but they are already too compiler-specific to remain as parser nodes. MLIR gives Tensorium a place to preserve this information while still reusing existing compiler infrastructure for canonicalization, common subexpression elimination, affine and SCF loops, memrefs, arithmetic operations, and LLVM lowering.

MLIR is not specific to Tensorium. It is already used or adopted in several compiler stacks where progressive lowering and custom dialects are important, including TensorFlow/XLA-style machine-learning compilation, the IREE compiler and runtime stack, and CIRCT for hardware design tooling [17, 18, 19]. Those projects use MLIR to keep domain structure visible until the compiler has done the domain-specific checks and rewrites it needs. Tensorium applies the same principle to numerical-relativity kernels: tensor variance, index roles, metric-derived fields, and kernel ABI metadata remain explicit until the point where scalar loops and memory descriptors are actually needed.

In Tensorium, MLIR is not just a generic output format. It is where the compiler defines domain operations, verifies structural invariants, attaches index-analysis metadata, and controls the boundary between geometric meaning and executable kernels. This gives Tensorium a tensor-aware source language backed by a compiler-visible IR stack for geometric kernels.

## 3 Language overview

The design of `Tensorium_lang` comes from a practical difficulty encountered while developing relativistic numerical solvers from scratch. The global software architecture of such codes is often a matter of engineering; the harder problem is implementing and checking the mathematical formalism correctly. Tensorial expressions, index rules, geometric identities, and formulation-dependent constraints are usually known to the developer, but remain implicit in the programming language. As a result, a lot of development time goes into redundant tests, manual algebra

checks, and making sure the implemented equations still match the intended mathematical system.

`Tensorium_lang` reduces this gap by making tensorial structure explicit at the language level. Instead of treating all quantities as raw arrays, fields are declared according to their geometric role: scalars, covariant tensors, contravariant tensors, spacetime metrics, inverse metrics, and evolution variables. Equations are written using indexed notation, and the compiler can analyze contractions, index consistency, metric-dependent operations, and geometric constructs before lowering the program to numerical kernels.

A program is organized around a small number of high-level blocks. Field declarations introduce the tensorial objects manipulated by the system. A `simulation` block specifies the coordinate system, dimensionality, grid resolution, time integration method, and spatial discretization scheme. An `initial_data` block defines the geometric background or initial configuration, while an `evolution` block describes the equations to be advanced in time.

```
field metric g[mu,nu]
field inverse_metric gU[mu,nu]
field scalar alpha
field cov_tensor2 gamma[i,j]
field con_tensor2 gammaU[i,j]
field cov_tensor2 K[i,j]
```

Greek indices such as $\mu, \nu$ refer to spacetime components, while Latin indices such as $i, j$ refer to spatial components. Although the implementation is still experimental, this distinction is already useful for expressing relativistic systems in notation close to the mathematical formalism.

```
simulation {
  coordinates = spherical
  dimension = 3
  resolution = [32,32,32]

  time {
    dt = 0.001
    integrator = rk3
  }

  spatial {
    scheme = fd
    derivative = centered
    order = 4
  }
}
```

In this example, the system is described on a three-dimensional spherical grid using a third-order Runge–Kutta time integrator and fourth-order centered finite differences. These declarations are intentionally kept separate from the tensorial equations themselves, so that the mathematical content and the numerical discretization remain distinguishable at the source level.

## 4 Compiler-visible structure

The main technical point of the language is not that it provides a new spelling for arrays. It is that the compiler sees information that would otherwise disappear into storage conventions. In a hand-written kernel, `gamma[6*n+p]` is just an address calculation. In Tensorium, before lowering, the compiler knows that `gamma` is a covariant rank-two spatial tensor, that `gammaU` is contravariant, that `metric4` is a spacetime metric, and that `split_3p1` requests a lapse/spatial-metric decomposition.

The current compiler makes the following facts explicit:

- tensor rank and variance, encoded as the pair $(n_{\text{up}}, n_{\text{down}})$;
- index roles, including which indices are free and which are contracted;
- metric declarations and metric-derived $3 + 1$ fields;
- differential operators before they are lowered to stencils;
- coordinate-system and discretization metadata;
- kernel buffer roles, tensor component counts, and read/write access in the generated ABI.

This information gives the compiler hooks for checks that ordinary array code cannot express directly. It can reject inconsistent index expressions, preserve metric operations until a dedicated lowering pass, and emit host descriptors that describe the geometric role of each generated buffer. The current implementation does not yet use all of this information for large symbolic simplifications. Its value right now is preservation, validation, and structured lowering of geometric kernels.

### 4.1 Einstein notation and `einsum` lowering

A main reason to keep tensorial structure in the compiler is the handling of Einstein summation. In Tensorium source, contractions are written in indexed notation:

```
dt A[i,j] = contract(B[i,k] * C[k,j])
```

The output indices `i,j` are free, while `k` is a contracted index. A conventional implementation would usually lower this expression immediately to nested loops over component arrays. Tensorium instead preserves the index structure through the backend IR as a `ContractionIR` and then lowers it to a dedicated MLIR operation:

```
tensorium.einsum ... {
  spec = "ik,kj->ij"
  tin.idx.ins = [["i", "k"], ["k", "j"]]
  tin.idx.out = ["i", "j"]
  tin.idx.roles = {i = "free", j = "free", k = "contracted"}
  tin.idx.valid = true
}
```

The concrete textual form of the operation is an implementation detail, but the metadata is the important part. The compiler records input index lists, output indices, occurrence counts, and roles. The Einstein analysis passes can then reject dangling indices, repeated output indices, invalid contractions, or summation patterns that are not allowed by the current lowering path.

This is an important difference from plain numerical code generation. Tensorium does more than translate tensor-looking syntax into loops: it keeps free and contracted indices visible so MLIR passes and generated ABI metadata can reason about them. This is also the mechanism behind the variance check below: illegal contractions can be diagnosed before component layout and grid loops are selected.

### 4.2 A concrete variance check

Consider the scalar contraction used in the Schwarzschild fixture:

```
dt H = contract(gammaU[i,j] * d_i(d_j(phi)))
```

Here `gammaU[i,j]` is contravariant and `d_i(d_j(phi))` is covariant, so the repeated indices can be contracted to a scalar. If a user accidentally writes

```
dt H = contract(gamma[i,j] * d_i(d_j(phi)))
```

then the expression contains only covariant indices. There is no legal upper/lower pair to contract. A raw C or C++ implementation would typically see two component arrays with compatible extents and would execute the wrong loop. Tensorium can reject the expression before lowering because the variance of `gamma` and `gammaU` is part of the source type and remains visible in the IR. This is the kind of compiler-visible information the project is designed to preserve.

# 5 Type system and source representation

The central object manipulated by the language is a typed tensor field. A field declaration carries both storage information and geometric intent. For example, `cov_tensor2 gamma[i,j]` denotes a covariant spatial rank-two tensor, whereas `metric g[mu,nu]` and `inverse_metric gU[mu,nu]` denote spacetime metric quantities. This distinction allows the compiler to reject inconsistent index expressions and to select different lowering paths for scalar, vector, tensor, and metric operations.

At the source level, the implementation uses the following tensor kinds:

```
Scalar, Vector, Covector,
CovTensor2, ConTensor2,
CovTensor3, ConTensor3,
CovTensor4, ConTensor4,
MixedTensor, Metric, InverseMetric
```

These kinds are normalized internally to a pair $(n_{\text{up}}, n_{\text{down}})$. For instance, a scalar has type $(0, 0)$, a covector has type $(0, 1)$, `cov_tensor2` has type $(0, 2)$, and `con_tensor2` has type $(2, 0)$.

The source abstract syntax tree distinguishes four classes of top-level objects: declarations, simulation metadata, initial data, and evolution systems. Expressions are represented as numeric literals, variables, binary operators, function calls, and indexed variables. During semantic analysis, indexed expressions receive inferred tensor types, and operations such as contraction or differentiation are checked against variance and rank rules.

The AST intentionally preserves source-level constructs that are meaningful to the compiler. A `Metric4InitDecl` is not immediately flattened into ordinary assignments, because later passes need to know that the 16 scalar components belong to a spacetime metric. Likewise, a `Split3P1BindingDecl` records the requested mapping from a spacetime metric to the lapse, shift, spatial metric, and inverse spatial metric. The main source objects currently include:

```
ExternDecl, FieldDecl, MetricDecl,
Metric4InitDecl, DecomposedMetricInitDecl,
Split3P1BindingDecl, InitialDataDecl,
EvolutionEq, EvolutionDecl, PrintDecl,
SimulationConfig, Program
```

Simulation metadata is also typed before lowering. The frontend normalizes coordinate systems, time integrators, spatial schemes, and derivative schemes to enumerated values:

```
CoordinateSystem = Cartesian | Spherical | Cylindrical
TimeIntegrator  = Euler | RK3 | RK4
SpatialScheme   = FiniteDifference | Spectral
DerivativeScheme = Centered | Upwind
```

These choices are later copied into the backend IR and MLIR module attributes, so that generated kernels and generated host descriptors agree about grid layout and derivative conventions.

# 6 Derived geometric quantities

Tensorium also exposes a compiler path for geometric quantities derived from a metric. The current implementation supports a built-in Christoffel construction from a spatial metric $\gamma_{ij}$ and its inverse $\gamma^{ij}$. For a torsion-free Levi–Civita connection, the reference formula is

$$\Gamma^{i}{}_{jk} = \frac{1}{2}\gamma^{i\ell}\left(\partial_j\gamma_{k\ell} + \partial_k\gamma_{j\ell} - \partial_\ell\gamma_{jk}\right). \tag{1}$$

In Tensorium source this construction is written as:

```
field cov_tensor2 gamma[i,j]
field con_tensor2 gammaU[i,j]
field mixed_tensor(up=1,down=2) Christoffel[i,j,k]

evolution MetricConnection {
  dt Christoffel[i,j,k] = christoffel(gamma, gammaU)
}
```

The same connection is used to define covariant derivatives. For a vector $V^i$, a covector $\omega_i$, and a mixed tensor $A^i{}_j$,

$$\nabla_k V^i = \partial_k V^i + \Gamma^{i}{}_{k\ell}V^\ell, \tag{2}$$

$$\nabla_k \omega_i = \partial_k \omega_i - \Gamma^{\ell}{}_{ki}\omega_\ell, \tag{3}$$

$$\nabla_k A^i{}_j = \partial_k A^i{}_j + \Gamma^{i}{}_{k\ell}A^\ell{}_j - \Gamma^{\ell}{}_{kj}A^i{}_\ell. \tag{4}$$

The sign of each connection term follows the variance of the corresponding index. This is another place where variance is compiler-visible rather than a storage convention.

Once the connection has been materialized, the spatial Ricci tensor can be constructed from the standard contraction of the Riemann tensor:

$$R_{ij} = \partial_k\Gamma^{k}{}_{ij} - \partial_j\Gamma^{k}{}_{ik} + \Gamma^{k}{}_{ij}\Gamma^{\ell}{}_{k\ell} - \Gamma^{k}{}_{i\ell}\Gamma^{\ell}{}_{jk}. \tag{5}$$

Tensorium expresses this formula directly in indexed notation:

```
evolution MetricRicci {
  Christoffel[i,j,k] = christoffel(gamma, gammaU)
  dt Ricci[i,j] =
      contract(d_k(Christoffel[k,i,j]))
    - contract(d_j(Christoffel[k,i,k]))
    + contract(Christoffel[k,i,j] * Christoffel[l,k,l])
    - contract(Christoffel[k,i,l] * Christoffel[l,j,k])
}
```

This path is not hard-coded to the Schwarzschild metric. It is written in terms of a covariant rank-two metric field and its inverse, so any spatial metric supplied to the compiler through the supported metric initialization path can feed the same Christoffel and Ricci construction. The current regression suite validates this path on the Schwarzschild spatial metric; the analytic BSSN fixture discussed later still injects an analytic Ricci tensor for that particular test surface.

# 7 Schwarzschild reference fixture

The reference fixture used throughout this note is the Schwarzschild initial data test in spherical coordinates. It is intentionally small enough to expose the complete compiler path while still exercising metric construction, $3+1$ decomposition, tensor field assignment, and an evolution block.

```
field metric g[mu,nu]
field inverse_metric gU[mu,nu]
field scalar alpha
field scalar phi
field scalar H
field cov_tensor2 gamma[i,j]
field con_tensor2 gammaU[i,j]
field cov_tensor2 K[i,j]

params { M }

initial_data {
  enforce_symmetry = true
  metric4 g[mu,nu] = [
    [-(1 - 2*M/r), 0, 0, 0],
    [0, 1/(1 - 2*M/r), 0, 0],
    [0, 0, r*r, 0],
    [0, 0, 0, r*r*sin(theta)*sin(theta)]
  ]
  split_3p1 {
    alpha -> alpha;
    gamma -> gamma[i,j];
    gammaU -> gammaU[i,j];
  }
}
```

For $f(r) = 1 - 2M/r$, the metric is

$$g_{\mu\nu} = \mathrm{diag}\left(-f, f^{-1}, r^2, r^2 \sin^2\theta\right). \tag{6}$$

The expected $3 + 1$ fields are

$$\alpha = \sqrt{f}, \qquad \gamma_{ij} = \mathrm{diag}\left(f^{-1}, r^2, r^2 \sin^2\theta\right), \qquad \gamma^{ij} = \mathrm{diag}\left(f, r^{-2}, (r^2 \sin^2\theta)^{-1}\right). \tag{7}$$

At the regression point $M = 1$, $r = 10$, and $\theta = \pi/2$, the test expects $f = 0.8$, $\alpha = \sqrt{0.8}$,

$$\gamma_{ij} = \mathrm{diag}(1.25, 100, 100), \qquad \gamma^{ij} = \mathrm{diag}(0.8, 0.01, 0.01). \tag{8}$$

These values are checked both by front-end evaluators and by LLVM-generated kernels linked with small C/C++ runtime runners.

# 8 Compiler pipeline

`Tensorium_lang` is implemented as a staged compiler. Each stage removes one layer of source-level notation while preserving enough structure for later validation and lowering. The frontend first builds a Tensorium-specific C++ IR because that representation is convenient for semantic checks and unit tests. MLIR then takes over at the point where Tensorium needs dialect operations, compiler passes, loop lowering, and LLVM-compatible code generation. The current pipeline is:

1. lexical analysis and parsing into a source AST;
2. semantic analysis and tensor type inference;
3. lowering to the Tensorium backend IR;
4. differential and Einstein-expression canonicalization;

5. lowering to the Tensorium MLIR dialect;
6. Tensorium-specific MLIR passes;
7. lowering through standard MLIR dialects to LLVM-compatible IR;
8. host ABI emission for generated kernels.

## 8.1 Public compiler interfaces

The public C++ API wraps the frontend, validator, and MLIR/LLVM emission path. It is intended for tools that need compiler functionality without shelling out to `Tensorium_cc`. The main entry points are:

```
parseAndValidateSource
parseAndValidateFile
emitMLIR
emitLLVMIR
compileSourceToLLVMIR
compileFileToLLVMIR
```

The high-level API accepts a `CompileOptions` object. The most important choice is the compilation mode. Executable mode enforces the constraints needed for code generation; symbolic mode allows a larger class of expressions to survive frontend analysis for inspection or future symbolic passes.

MLIR generation is controlled by `MLIRGenOptions` and `MLIRPassOptions`. These options expose optimization level, diagnostic settings, pass timing, best-effort lowering, canonicalization, CSE, inlining, metric lowering, initial-data lowering, grid-kernel lowering, Einstein lowering, stencil lowering, and optional dissipation. The command-line driver is a thin orchestration layer over the same option structure.

## 8.2 Source AST and semantic layer

The parser constructs a `Program` object containing parameter declarations, extern declarations, field declarations, metrics, evolutions, prints, simulation metadata, and initial data. Expressions remain close to the surface syntax: numeric constants, variable references, binary operators, calls, parentheses, and indexed variables.

The semantic layer assigns tensor information to indexed expressions. The important invariant is that all expressions have an inferred rank and variance before they reach the backend IR. A contraction must remove one covariant and one contravariant index; a partial derivative adds one covariant derivative index; and covariant derivative expressions require metric or connection information depending on the requested form. The compiler therefore rejects many index errors before any grid or memory layout has been selected.

The compiler also distinguishes two modes. In executable mode, unknown scalar calls are rejected unless declared through the extern mechanism, because the lowering path must produce executable code. In symbolic mode, more expressions can be retained for analysis, but they are not necessarily lowerable to LLVM.

## 8.3 Backend IR

After semantic analysis, the program is lowered into a C++ backend IR. This IR is independent of MLIR and acts as the compiler's typed domain model. The main module object contains:

- `SimulationIR`: coordinate system, dimension, resolution, time integrator, and spatial derivative scheme;

- `FieldIR`: field name, field kind, and tensor type $(n_{\text{up}}, n_{\text{down}})$;
- `InitialDataIR`: either an explicit spacetime metric `Metric4InitIR` or decomposed $3+1$ data, plus optional `split_3p1` bindings;
- `EvolutionIR`: time-derivative assignments and temporary tensor assignments.

Expression nodes in this IR include scalar operators and the following tensor-specific nodes:

```
TensorProductIR
ContractionIR
IndexRenameIR
IndexPermuteIR
TraceIR
PartialDerivativeIR
GradientIR
CovariantDerivativeIR
DivergenceIR
```

This layer is the first point where differential operations and Einstein notation are normalized independently of the source syntax.

The choice to keep this IR outside MLIR is deliberate. It gives the frontend a small, C++-native representation for tensor-specific validation and makes unit tests independent of MLIR context construction. It also keeps Tensorium's geometric concepts explicit before they are forced into lower-level operations. For example, a `CovariantDerivativeIR` can be checked as a geometric operation before a later pass decides whether it should become an indexed Einstein expression, a stencil, or a symbolic placeholder.

Two validation passes run on this representation. The differential canonicalizer rewrites derivative constructs into a consistent internal form. The Einstein canonicalizer normalizes contraction and index-ordering patterns before they are emitted as `tensorium.einsum`. The backend verifier then checks the resulting IR before MLIR generation.

### 8.4 Tensorium MLIR dialect

The MLIR layer introduces a Tensorium dialect. The central MLIR type is

```
!tensorium.field<f64, up, down>
```

where `up` and `down` encode tensor variance. Thus the Schwarzschild lapse has type $(0, 0)$, the spatial metric has type $(0, 2)$, and the inverse spatial metric has type $(2, 0)$.

The dialect contains scalar operations and tensor-specific operations:

```
tensorium.const      tensorium.param
tensorium.coord      tensorium.add
tensorium.sub        tensorium.mul
tensorium.div        tensorium.sin
tensorium.sqrt       tensorium.ref
tensorium.deriv      tensorium.contract
tensorium.promote    tensorium.dt_assign
tensorium.einsum
```

Among these operations, `tensorium.einsum` is the bridge between indexed source equations and executable tensor contractions. It carries an einsum specification and index-analysis attributes such as `tin.idx.ins`, `tin.idx.out`, `tin.idx.counts`, `tin.idx.roles`, and `tin.idx.valid`. These attributes are deliberately retained in the Tensorium dialect so that later passes can verify the contraction before scalar loop lowering erases the distinction between free and summed indices.

Metric and initial-data constructs are represented explicitly:

- `tensorium.metric4` carries 16 scalar components of a spacetime metric and metadata such as coordinate system, index names, and symmetry enforcement;
- `tensorium.decompose3p1_from_metric` produces $(\alpha, \beta_i, \gamma_{ij}, \gamma^{ij})$ from a spacetime metric;
- `tensorium.init3p1` carries explicit or decomposed initial fields;
- tensor builders materialize tensor values from scalar components.

```
build_covector
build_cov_tensor2
build_con_tensor2
```

The dialect uses MLIR traits and verifiers to encode technical invariants. For example, `metric4` must carry exactly 16 components, `build_cov_tensor2` and `build_con_tensor2` must carry 9 components, multiplication increases rank additively, derivative operations add one covariant index, and contraction cannot increase rank.

For the Schwarzschild fixture, the initial-data MLIR contains a `tensorium.metric4` operation followed by `tensorium.decompose3p1_from_metric`, `tensorium.init3p1`, and three `tensorium.assign` operations writing `alpha`, `gamma`, and `gammaU`. This is the compiler-level representation of the source `split_3p1` block.

Evolution blocks are lowered to executable RHS kernels. In the current pipeline, a Tensorium `evolution` declaration can produce grid-level functions such as `tensorium_rhs_grid_affine`, which evaluate time-derivative outputs over a grid. These kernels are suitable for use inside an explicit update loop, but the loop itself is still supplied by host-side C/C++ code in the current implementation.

### 8.5 Tensorium MLIR passes

The current pass set is deliberately explicit. Passes can be selected from the command line and are also enabled automatically when LLVM or host-header emission is requested. The relevant passes are:

- `tensorium-metric-lower`: lowers metric and $3+1$ constructs toward scalar and tensor component operations;
- initial-data standard lowering emits the point kernel `tensorium_init_point`;
- `tensorium-init-grid-affine-lower`: builds a grid-level affine loop kernel over coordinate and output buffers;
- `tensorium-rhs-grid-affine-lower`: builds grid-level RHS kernels for evolution equations;
- `tensorium-einstein-lower`: lowers symbolic contractions into `tensorium.einsum`;
- `tensorium-index-role-analysis`: annotates einsum operations with input indices, output indices, occurrence counts, roles, and validity;
- the Einstein analysis passes inspect and normalize summation forms: `analyze-einsum`, `canonicalize`, and `validate`;
- `tensorium-stencil-lower`: lowers derivative operations to finite-difference stencils according to the selected spacing and stencil order;
- `tensorium-dissipation`: optionally inserts Kreiss–Oliger dissipation into time-derivative assignments;

- `tensorium-strip-source-funcs`: removes source-level functions after generated grid kernels have been emitted.

The executable pipeline is ordered so that source-level geometric operations are expanded before source functions are removed. In the current implementation the safe Einstein path applies metric lowering, initial-data lowering, grid kernel generation, source-function stripping, Einstein lowering, index-role analysis, optional Einstein diagnostics, optional stencil lowering, and optional dissipation. Canonicalization and CSE are applied after Tensorium-specific rewrites to remove redundant scalar arithmetic introduced by metric and stencil expansion.

After Tensorium-specific lowering, the compiler runs standard MLIR normalization and LLVM conversion passes: canonicalization, common subexpression elimination, affine lowering, SCF-to-CF conversion, memref metadata expansion, arithmetic/math/index conversion, func-to-LLVM conversion, memref-to-LLVM finalization, and unrealized-cast reconciliation.

## 8.6 Generated ABI and host interface

Generated modules carry ABI attributes. The current ABI version is 1 and uses a component-major structure-of-arrays layout:

```
tensorium.abi.version = 1
tensorium.abi.memory_layout = "soa_component_major"
tensorium.abi.memref_abi = "strided_memref_rank1_f64"
```

The generated symbols distinguish source-level entry points from point and grid kernels for initialization and RHS evaluation. The exact symbol list is an implementation detail; the important property is that each exported kernel is paired with ABI metadata describing argument kinds, buffer roles, and access modes.

The C ABI uses rank-one strided memref descriptors for floating-point buffers:

```
struct StridedMemRef1DF64 {
  double *allocated;
  double *aligned;
  int64_t offset;
  int64_t size;
  int64_t stride;
};
```

Host metadata is represented by `HostModuleABI` and `HostKernelABI`. The metadata records the coordinate system, dimension, resolution, spatial scheme, derivative order, fields, output names, tensor component counts, kernel symbols, raw argument kinds, buffer roles, and read-/write argument sets. Development runners use the generated host header to call the lowered kernels directly and compare generated values against analytic expectations.

## 8.7 Runtime storage model

The runtime layer provides host-side storage objects that mirror the generated ABI. Two classes are central:

```
HostFieldStorage
GeneratedHostStorage
```

`HostFieldStorage` is used with compiler-side ABI metadata, while `GeneratedHostStorage` consumes descriptor arrays emitted into generated headers. Both models allocate one contiguous arena and expose named logical buffers for coordinates, fields, and outputs.

Each logical buffer records its role, scalar count, tensor component count, and offset into the component-major structure-of-arrays arena. Kernel binding plans then map generated function

arguments to these buffers with explicit read, write, or read-write access. The runtime therefore does not infer calling conventions from C++ names; it follows the ABI descriptors emitted by the compiler. This is essential for regression tests, because the same generated kernel can be called by a small development runner or by a larger host program without changing the generated LLVM IR.

The generated storage layer also exposes simple Euler update metadata for RHS kernels. That mechanism is intentionally minimal: it exists to test that generated RHS outputs can be paired with evolved fields, not to define a full time-integration framework.

Thus, Tensorium is currently a RHS-kernel generator rather than a complete solver generator. It can emit initialization and RHS kernels with a stable host ABI, but a complete simulation still requires external C, C++, or Fortran code to allocate grids, call kernels, apply time integrators, manage boundary conditions, and orchestrate a full evolution.

### 8.8 Driver modes and diagnostics

The `Tensorium_cc` driver exposes the compiler pipeline directly. It can dump the source AST, indexed expressions, backend IR, Tensorium MLIR, LLVM IR, and generated host headers. It also exposes pass selection flags, stencil spacing and order, dissipation strength, MLIR pass timing, best-effort lowering, and toggles for canonicalization, CSE, and inlining.

The driver defaults are intentionally conservative. When executable LLVM or a host header is requested, the driver enables the lowering stages required for the current runtime path: metric lowering, initial-data lowering, grid-kernel lowering, RHS grid lowering, and source-function stripping. More specialized analysis passes remain explicit so that tests can isolate the pass under inspection.

### 8.9 Executable command path

The executable path produces LLVM IR plus a host header. Development tests then compile the LLVM IR to an object file, link it with a small runner, and compare the generated numerical values against analytic expectations. A concrete command line for the full Schwarzschild BSSN fixture is given in the appendix.

## 9 Validation tests

The validation suite is designed to test compiler behavior through explicit geometric examples. Simple scalar and tensor programs check parsing, semantic analysis, index validation, and lowering. General relativity fixtures then test the representation of standard metrics and the construction of derived quantities.

The tests are organized by compiler layer:

- frontend tests check tokenization, parser diagnostics, AST shape, semantic errors, unknown extern calls, and tensor declaration rules;
- backend IR tests check tensor type propagation, differential canonicalization, Einstein-expression canonicalization, and verifier failures;
- MLIR tests check emitted dialect structure, operation attributes, pass-specific rewrites, and generated ABI metadata;
- evaluator tests execute the frontend numeric evaluator for the subset of initial-data operations needed by the Schwarzschild milestone;
- LLVM smoke tests compile emitted LLVM IR, link it with small C or C++ runners, and compare printed numerical values;

- runtime tests exercise generated host descriptors, buffer allocation, kernel binding plans, and simple Euler update wiring.

The non-trivial compiler checks are not only the final floating-point equalities. They include index-variance rejection, component-count checks for metric and tensor builders, preservation of metric operations until metric lowering, and ABI checks that distinguish coordinate buffers, input fields, output fields, and write-only RHS buffers. The analytic Schwarzschild tests are therefore intentionally simple as physics tests, but useful as compiler regressions: a change that swaps `gamma` for `gammaU`, drops a metric component, changes a tensor variance, or mislabels an output buffer can be caught before or during lowering even when the numerical solution being sampled is static.

The Schwarzschild tests validate a diagonal spherical metric, its spatial metric, inverse spatial metric, lapse, Christoffel symbols, Ricci quantities, Hamiltonian constraint fixtures, and BSSN-related reductions. For the Schwarzschild initial-data case, tests check both the symbolic MLIR structure and the generated numeric kernel output. At the reference point, the runner prints and checks:

```
alpha      = 0.89442719099991586
gamma      = diag(1.25, 100, 100)
gammaU     = diag(0.8, 0.01, 0.01)
gammaU*gamma = identity
```

Additional fixtures cover Reissner–Nordstrom, Minkowski, spatial off-diagonal metrics, Kerr-like shift terms, covariant derivatives, and runtime BSSN smoke tests. These tests are not intended as a replacement for a full numerical relativity benchmark suite; they are regression tests for the compiler's geometric lowering behavior.

## 10 Implementation details

The implementation is organized around explicit representation boundaries. The frontend modules perform lexing, parsing, AST construction, semantic analysis, and lowering to a C++ Tensorium IR. The MLIR modules define the Tensorium dialect, dialect verifiers, Tensorium-specific lowering passes, LLVM emission, host-header emission, and generated ABI descriptors. The runtime modules provide host-side buffer storage and generated-kernel binding plans.

These details are intentionally secondary to the paper's main claim. The important design choice is not the repository layout, but the fact that each stage receives a representation that still exposes tensor rank, variance, metric operations, and kernel interface metadata. The command-line driver and development scripts are engineering scaffolding around that pipeline: they parse fixtures, emit MLIR or LLVM IR, generate host headers, compile smoke runners, and compare generated values against analytic expectations.

## 11 Limitations and future work

The current implementation is intentionally limited. The language can express and lower useful tensorial fixtures, but it is not yet a complete symbolic system for arbitrary relativistic field equations. In particular, the current Schwarzschild tests validate the metric, 3+1 decomposition, generated initial-data kernels, Christoffel/Ricci-related fixtures, and selected BSSN checks, but they do not yet constitute a full end-to-end numerical relativity application.

The analytic BSSN fixture should be interpreted in that context. It checks that an encoded Schwarzschild BSSN constraint surface is preserved through the compiler and executed correctly. It does not yet demonstrate automatic construction of the BSSN equations, automatic Ricci-tensor derivation for an arbitrary metric, or a production evolution code.

This limitation is specific to the BSSN fixture, not to the lower-level Ricci building blocks. Tensorium already has a metric-derived path for $\Gamma^i{}_{jk}$ and $R_{ij}$ from $\gamma_{ij}$ and $\gamma^{ij}$. What is not yet implemented is the automatic assembly of an entire BSSN formulation, including all geometric source terms and gauge equations, from only a high-level formulation declaration.

Even with that limit, the executable pipeline can already generate RHS kernels that can be evolved by host code. The missing piece is the surrounding solver: time-step orchestration, boundary handling, grid hierarchy management, I/O, and production runtime integration still live outside Tensorium, in C/C++ or Fortran host programs.

The next steps are clear. First, scalar functions such as $h(r)$, $f(r)$, and $\omega(r)$ should become first-class symbolic objects in initial data, rather than externally supplied pointwise parameters. Second, linearization with respect to a chosen symbol should become a general compiler mode instead of a test-specific convention. Third, the compiler should expose more geometric simplification passes, allowing tests such as $d(r^4\omega')/dr = 0$ to be represented directly in the DSL. Fourth, the generated kernels should be connected to a real adaptive-mesh runtime and the host-side code generation should become nearly complete. A likely target is an AMReX-based backend [20], so that Tensorium can generate most of the C/C++ glue for block management, ghost zones, refinement, distributed memory execution, kernel launches, and explicit update loops. Finally, the language should be evaluated on larger evolution systems and compared against established numerical-relativity workflows.

Modified-gravity systems and higher-level formulation-specific front ends are future work. They should be built only after the lower-level tensor, metric, ABI, and symbolic infrastructure is stable.

## 12 Conclusion

`Tensorium_lang` explores a compiler-first approach to tensorial systems in numerical relativity. The main idea is simple: tensorial structure should be visible to the compiler, not hidden in array layout conventions. By giving the language explicit notions of tensor variance, indexed expressions, spacetime metrics, $3+1$ decomposition, and kernel ABI metadata, the compiler can check and preserve information that ordinary low-level kernels discard before optimization begins.

The current implementation has an end-to-end path from source notation to executable kernels. A program is parsed and typed, lowered to a Tensorium IR, emitted into a Tensorium MLIR dialect, transformed through metric and grid-kernel lowering passes, converted to LLVM-compatible IR, and exposed through a generated host ABI. In particular, Tensorium can already produce RHS kernels for `evolution` blocks and those kernels can be called from external host code. The Schwarzschild and analytic BSSN fixtures show that this path can preserve metric and tensor structure through code generation and runtime execution.

The compiler is also beginning to own derived geometry. Christoffel symbols can be generated from a metric and inverse metric, covariant derivatives use the resulting connection with variance-aware signs, and the spatial Ricci tensor can be expressed and lowered from the Christoffel contraction formula. This separates the current compiler path from a plain array-kernel generator: the metric is not only data, but a source for further geometric objects.

The result should be read with this scope in mind. Tensorium is not yet a full symbolic differential-geometry engine, and the BSSN regression does not derive the complete BSSN system automatically. It validates an encoded analytic surface and the compiler infrastructure around it. This is still a useful step: it exercises tensor variance, indexed contractions, metric initialization, $3+1$ decomposition, MLIR lowering, generated ABI descriptors, and runtime kernel calls in one compiler path. A complete solver, however, still requires host-side C/C++ or Fortran code around the generated kernels.

The longer-term direction is to move more of the geometry into the compiler: automatic

curvature construction, stronger symbolic simplification, linearization modes, formulation-level checks, and integration with an adaptive-mesh runtime such as AMReX. The intended direction is for Tensorium to generate not only geometric RHS kernels, but also most of the C/C++ host glue needed to run them inside a real AMR solver. If those pieces are completed, Tensorium could connect relativistic formulations to low-level performance kernels while keeping the mathematical structure visible at each compiler stage.

## References


[1] R. Arnowitt, S. Deser, and C. W. Misner. The dynamics of general relativity. In L. Witten, editor, *Gravitation: An Introduction to Current Research*, pages 227–265. Wiley, New York, 1962.

[2] M. Alcubierre. *Introduction to 3+1 Numerical Relativity*. Oxford University Press, Oxford, 2008. doi:10.1093/acprof:oso/9780199205677.001.0001.

[3] E. Gourgoulhon. *3+1 Formalism in General Relativity: Bases of Numerical Relativity*. Lecture Notes in Physics, Vol. 846. Springer, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-24525-1.

[4] M. Shibata and T. Nakamura. Evolution of three-dimensional gravitational waves: Harmonic slicing case. *Physical Review D*, 52:5428–5444, 1995. doi:10.1103/PhysRevD.52.5428.

[5] T. W. Baumgarte and S. L. Shapiro. On the numerical integration of Einstein's field equations. *Physical Review D*, 59:024007, 1998. doi:10.1103/PhysRevD.59.024007.

[6] T. W. Baumgarte and S. L. Shapiro. *Numerical Relativity: Solving Einstein's Equations on the Computer*. Cambridge University Press, Cambridge, 2010.

[7] S. Husa, I. Hinder, and C. Lechner. Kranc: A Mathematica application to generate numerical codes for tensorial evolution equations. *Computer Physics Communications*, 174(12):983–1004, 2006. doi:10.1016/j.cpc.2006.02.002.

[8] I. Ruchlin, Z. B. Etienne, and T. W. Baumgarte. SENR/NRPy+: Numerical relativity in singular curvilinear coordinate systems. *Physical Review D*, 97:064036, 2018. doi:10.1103/PhysRevD.97.064036.

[9] F. Löffler, J. Faber, E. Bentivegna, T. Bode, P. Diener, R. Haas, I. Hinder, B. C. Mundim, C. D. Ott, E. Schnetter, G. Allen, M. Campanelli, and P. Laguna. The Einstein Toolkit: A community computational infrastructure for relativistic astrophysics. *Classical and Quantum Gravity*, 29(11):115001, 2012. doi:10.1088/0264-9381/29/11/115001.

[10] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR 2002*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2003.

[11] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. SymPy: Symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017. doi:10.7717/peerj-cs.103.

[12] N. Kukreja, M. Louboutin, F. Vieira, F. Luporini, M. Lange, and G. Gorman. Devito: Automated fast finite difference computation. In *Proceedings of the 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC 2016, 2016. doi:10.1109/WOLFHPC.2016.06.

[13] A. Logg, K.-A. Mardal, and G. N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer, Berlin, Heidelberg, 2012.

[14] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):24, 2017. doi:10.1145/2998441.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO 2004, pages 75–88, 2004. doi:10.1109/CGO.2004.1281665.

[16] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2021, pages 2–14, 2021. doi:10.1109/CGO51591.2021.9370308.

[17] TensorFlow authors. TensorFlow MLIR. `https://www.tensorflow.org/mlir`. Accessed 2026-05-23.

[18] IREE authors. IREE: Intermediate Representation Execution Environment. `https://iree.dev/`. Accessed 2026-05-23.

[19] CIRCT authors. CIRCT: Circuit IR Compilers and Tools. `https://circt.llvm.org/`. Accessed 2026-05-23.

[20] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale. AMReX: A framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370, 2019. doi:10.21105/joss.01370.

# A Complete Schwarzschild BSSN compiler pipeline

This appendix records the complete compiler path for the current Schwarzschild BSSN constraint fixture. Unlike the small metric-only example, this fixture carries the analytic BSSN state, a non-trivial RHS surface, and the ADM constraints checked by the development runners.

## A.1 Input fixture

The source file is:

```
tests/fixtures/gr/schwarzschild_bssn_constraints_analytic_3d.tn
```

It starts from the Schwarzschild spatial metric on a time-symmetric slice,

$$\gamma_{ij} = \mathrm{diag}\left(f^{-1}, r^2, r^2 \sin^2\theta\right), \qquad f = 1 - \frac{2M}{r}. \tag{9}$$

The fixture declares the spacetime metric, the $3+1$ fields, the BSSN state, the analytic Ricci/Hessian auxiliaries, and the constraint outputs:

```
field scalar chi
field vector beta[i]
field vector B[i]
field scalar K
field cov_tensor2 gammatilde[i,j]
field con_tensor2 gammatildeU[i,j]
field cov_tensor2 Atilde[i,j]
field vector Gammahat[i]

field cov_tensor2 RicciAnalytic[i,j]
field cov_tensor2 HessianAlpha[i,j]
field cov_tensor3 DAtilde[i,j,k]
field scalar Hamiltonian
field covector Momentum[i]
```

The static BSSN reduction encoded by the test is:

```
chi = 1
beta^i = 0
B^i = 0
K = 0
gammatilde_ij = gamma_ij
gammatilde^ij = gamma^ij
Atilde_ij = 0
Gammahat^i = 0
```

## A.2 Reference BSSN system

The fixture is a regression test for a reduced analytic Schwarzschild case. It does not encode a generic BSSN evolution system term by term. The reference system behind the test, however, is the vacuum BSSN system [1, 2, 3, 4, 5, 6]. In the notation used here,

$$\gamma_{ij} = \chi^{-1}\tilde{\gamma}_{ij}, \qquad K_{ij} = \chi^{-1}\left(\tilde{A}_{ij} + \frac{1}{3}\tilde{\gamma}_{ij}K\right), \qquad \tilde{\Gamma}^i = -\partial_j\tilde{\gamma}^{ij}. \tag{10}$$

The determinant constraint is $\det\tilde{\gamma}_{ij} = 1$, and $\tilde{A}_{ij}$ is trace-free with respect to $\tilde{\gamma}^{ij}$.

With lapse $\alpha$, shift $\beta^i$, and $\phi = -\frac{1}{4}\log\chi$, the reference vacuum BSSN evolution system is:

$$\partial_t \chi = \beta^k \partial_k \chi + \frac{2}{3}\chi\left(\alpha K - \partial_k \beta^k\right), \tag{11}$$

$$\partial_t \tilde{\gamma}_{ij} = \beta^k \partial_k \tilde{\gamma}_{ij} + \tilde{\gamma}_{ik}\partial_j \beta^k + \tilde{\gamma}_{kj}\partial_i \beta^k - \frac{2}{3}\tilde{\gamma}_{ij}\partial_k \beta^k - 2\alpha \tilde{A}_{ij}, \tag{12}$$

$$\partial_t K = \beta^k \partial_k K - D^i D_i \alpha + \alpha\left(\tilde{A}_{ij}\tilde{A}^{ij} + \frac{1}{3}K^2\right), \tag{13}$$

$$\begin{aligned}\partial_t \tilde{A}_{ij} = {} & \beta^k \partial_k \tilde{A}_{ij} + \tilde{A}_{ik}\partial_j \beta^k + \tilde{A}_{kj}\partial_i \beta^k - \frac{2}{3}\tilde{A}_{ij}\partial_k \beta^k \\ & + \chi\left[-D_i D_j \alpha + \alpha R_{ij}\right]^{\mathrm{TF}} + \alpha\left(K\tilde{A}_{ij} - 2\tilde{A}_{ik}\tilde{A}^k{}_j\right),\end{aligned} \tag{14}$$

$$\begin{aligned}\partial_t \tilde{\Gamma}^i = {} & \beta^k \partial_k \tilde{\Gamma}^i - \tilde{\Gamma}^k \partial_k \beta^i + \frac{2}{3}\tilde{\Gamma}^i \partial_k \beta^k + \tilde{\gamma}^{jk}\partial_j \partial_k \beta^i + \frac{1}{3}\tilde{\gamma}^{ij}\partial_j \partial_k \beta^k \\ & - 2\tilde{A}^{ij}\partial_j \alpha + 2\alpha\left(\tilde{\Gamma}^i{}_{jk}\tilde{A}^{jk} - \frac{2}{3}\tilde{\gamma}^{ij}\partial_j K + 6\tilde{A}^{ij}\partial_j \phi\right).\end{aligned} \tag{15}$$

The gauge equations used as reference are the standard one-plus-log lapse and a simplified Gamma-driver shift surface:

$$\partial_t \alpha = \beta^k \partial_k \alpha - 2\alpha K, \tag{16}$$

$$\partial_t \beta^i = B^i, \tag{17}$$

$$\partial_t B^i = \frac{3}{4}\partial_t \tilde{\Gamma}^i - \eta B^i. \tag{18}$$

The corresponding vacuum constraints are:

$$\mathcal{H} = R + \frac{2}{3}K^2 - \tilde{A}_{ij}\tilde{A}^{ij} = 0, \tag{19}$$

$$\mathcal{M}_i = \tilde{D}_j \tilde{A}^j{}_i + 6\tilde{A}^j{}_i \partial_j \phi - \frac{2}{3}\partial_i K = 0. \tag{20}$$

The fixture specializes this system before it reaches the DSL. For the Schwarzschild time-symmetric slice used by the test, $\chi = 1$, $\beta^i = 0$, $B^i = 0$, $K = 0$, $\tilde{A}_{ij} = 0$, and $\tilde{\gamma}_{ij} = \gamma_{ij}$. Transport terms, shift terms, and Gamma-driver source terms therefore vanish. The exact reduced surfaces checked by the fixture are:

$$\partial_t \chi = \frac{2}{3}\alpha\chi K, \tag{21}$$

$$\partial_t \alpha = -2\alpha K, \tag{22}$$

$$\partial_t \beta^i = B^i, \tag{23}$$

$$\partial_t B^i = -\eta B^i, \tag{24}$$

$$\partial_t K = -D^i D_i \alpha + \alpha\left(R + \tilde{A}_{ij}\tilde{A}^{ij} + \frac{1}{3}K^2\right), \tag{25}$$

$$\partial_t \tilde{\gamma}_{ij} = -2\alpha \tilde{A}_{ij}, \tag{26}$$

$$\partial_t \tilde{A}_{ij} = \left[-D_i D_j \alpha + \alpha R_{ij}\right]^{\mathrm{TF}} + \alpha\left(K\tilde{A}_{ij} - 2\tilde{A}_{ik}\tilde{A}^k{}_j\right), \tag{27}$$

$$\partial_t \tilde{\Gamma}^i = 0. \tag{28}$$

The fixture also exports the diagnostic constraint surface it actually computes:

$$\mathcal{H}_{\mathrm{fixture}} = \chi\tilde{\gamma}^{ij}R_{ij} + K^2 - \tilde{A}_{ij}\tilde{A}^{ij}, \tag{29}$$

$$\mathcal{M}_i^{\mathrm{fixture}} = \gamma^{jk}\nabla_k \tilde{A}_{ij} - \frac{2}{3}\partial_i K. \tag{30}$$

On the encoded Schwarzschild data, $K = 0$, $\tilde{A}_{ij} = 0$, and $D_i D_j \alpha = \alpha R_{ij}$. Thus the full reference system and the reduced fixture agree on the values being tested: the BSSN RHS vanishes and both diagnostic constraints are zero. The fixture should therefore be read as a static analytic BSSN regression, not as a generic BSSN solver.

### A.3 Analytic equation surface

The fixture avoids deriving the full spatial Ricci tensor in the compiler. It instead encodes the analytic Schwarzschild Ricci tensor and the static-vacuum identity for the lapse Hessian:

```
f = 1.0 - 2.0*M/Rcoord

AnalyticRicci[i,j] =
    (M/(Rcoord*Rcoord*Rcoord)) * gamma[i,j]
  - ((3.0*M)/(Rcoord*Rcoord*Rcoord*f)) * radialBasis[i,j]

AnalyticHessianAlpha[i,j] = alpha * AnalyticRicci[i,j]
```

The reduced BSSN RHS surface is then expressed in indexed Tensorium notation:

```
dt dchi = (2.0/3.0) * alpha * chi * K + BetaZero
dt dalpha = -2.0 * alpha * K + BetaZero
dt dbeta[i] = B[i]
dt dB[i] = (3.0/4.0) * GammaDriver[i] - eta * B[i]

dt dK =
    -LaplacianAlpha
  + alpha * (RicciScalar + AtildeSq + (1.0/3.0) * K*K)

dt dgammatilde[i,j] =
    -2.0 * alpha * Atilde[i,j] + MetricZero[i,j]

dt dAtilde[i,j] =
    AtildeTraceFreeSource[i,j]
  - (1.0/3.0) * gammatilde[i,j] * AtildeTrace
  + alpha * K * Atilde[i,j]
  - 2.0 * alpha
      * contract(Atilde[i,k] * gammatildeU[k,l] * Atilde[l,j])

dt dGammahat[i] = GammaDriver[i]
```

The constraint outputs are part of the same evolution surface:

```
dt RicciAnalytic[i,j] = AnalyticRicci[i,j]
dt HessianAlpha[i,j] = AnalyticHessianAlpha[i,j]
dt DAtilde[i,j,k] = DAtildeAnalytic[i,j,k]

dt Hamiltonian =
    RicciScalar
  + K*K
  - AtildeSq

dt Momentum[i] =
    contract(gammaU[j,k] * DAtildeAnalytic[i,j,k])
  - (2.0/3.0) * d_i(K)
```

On the static Schwarzschild state this encoded analytic surface should produce zero RHS values for the BSSN state variables, zero Hamiltonian constraint, and zero momentum constraint. This validates preservation and execution of the encoded tensorial surface; it does not yet validate automatic derivation of $R_{ij}$ or of the full BSSN system from first principles.

## A.4 Frontend and backend IR

The parser constructs a `Program` with one `InitialDataDecl` and one `EvolutionDecl` named:

```
SchwarzschildBSSNFullConstraintsAnalytic
```

Semantic analysis assigns field variance before lowering:

```
chi, K, Hamiltonian     -> scalar
beta, B, Gammahat       -> vector
Momentum                -> covector
gamma, gammatilde       -> cov_tensor2
gammaU, gammatildeU     -> con_tensor2
Atilde, RicciAnalytic   -> cov_tensor2
HessianAlpha            -> cov_tensor2
DAtilde                 -> cov_tensor3
```

The backend IR preserves the important tensor operations explicitly:

```
TensorProductIR
ContractionIR
PartialDerivativeIR
CovariantDerivativeIR
```

This is the level at which expressions such as the momentum contraction remain geometric operations rather than loops over component arrays:

```
contract(gammaU[j,k] * DAtildeAnalytic[i,j,k])
```

## A.5 MLIR and lowering

MLIR generation emits an initial-data function for the Schwarzschild metric and a RHS function for the analytic BSSN surface:

```
func.func @tensorium_init(...)
  tensorium.metric4 ...
  tensorium.decompose3p1_from_metric ...
  tensorium.assign "alpha"
  tensorium.assign "gamma"
  tensorium.assign "gammaU"

func.func @tensorium_rhs(...)
  tensorium.ref "gammatilde"
  tensorium.ref "gammatildeU"
  tensorium.ref "Atilde"
  tensorium.ref "Rcoord"
  tensorium.ref "radialBasis"
  tensorium.contract ...
  tensorium.deriv ...
  tensorium.dt_assign "Hamiltonian"
  tensorium.dt_assign "Momentum"
```

The development script uses the actual executable lowering path:

```
Tensorium_cc \
  --tensorium-metric-lower \
  --tensorium-init-std-lower \
  --tensorium-init-grid-affine-lower \
  --tensorium-rhs-grid-affine-lower \
  --tensorium-stencil-lower \
  --tensorium-strip-source-funcs \
  --emit-llvm tensorium_schwarzschild_bssn_constraints.ll \
  --emit-host-header tensorium_schwarzschild_bssn_constraints_host.h \
  tests/fixtures/gr/schwarzschild_bssn_constraints_analytic_3d.tn
```

The generated module exposes at least the following callable kernels:

```
tensorium_init_grid_affine
tensorium_rhs_grid_affine
```

### A.6 Runtime regression

The primary low-level runner is:

```
tools/dev/test_schwarzschild_bssn_constraints_ll.sh
```

It compiles the emitted LLVM IR, includes the generated host header, links `tools/dev/ll_rhs_runner_schwarzschild_bssn_constraints.c`, and executes the generated kernels. The runner fills the coordinate buffers, calls `tensorium_call_init_grid_affine`, copies the generated `gamma` and `gammaU` into the BSSN conformal metric fields, fills `Rcoord` and `radialBasis`, and then calls `tensorium_call_rhs_grid_affine`.

The checks cover the center point and all interior grid points:

```
alpha == sqrt(1 - 2*M/r)
gammaU * gamma == identity
chi == 1
beta == B == Gammahat == 0
K == 0
gammatilde == gamma
gammatildeU == gammaU
Atilde == 0
dchi, dalpha, dbeta, dB, dK == 0
dgammatilde, dAtilde, dGammahat == 0
RicciAnalytic == analytic Schwarzschild Ricci
HessianAlpha == alpha * RicciAnalytic
DAtilde == 0
Hamiltonian == 0
Momentum == 0
```

A second runtime test exercises the generated host-storage descriptors:

```
tools/dev/test_runtime_uniform_schwarzschild_bssn.sh
```

That test uses `GeneratedHostStorage`, generated buffer descriptors, kernel binding plans, and Euler-update metadata. It validates the same Schwarzschild BSSN constraint surface through the generated ABI rather than through manually assembled C arrays.

This gives the complete compiler-level contract for the Schwarzschild BSSN case: the source metric and indexed BSSN equations are parsed, semantically typed, preserved as backend tensor operations, emitted as Tensorium MLIR, lowered to grid kernels, exposed through the generated ABI, and checked by runtime runners against the analytic static-vacuum solution.