AD FALCON API Manual

UMAT Authoring Guide

Javad Ghorbani

March 14, 2026

Contents

1 UMAT Authoring Guide

This guide walks external developers through building, packaging, and plugging a custom User MATerial (UMAT) into FALCON. It assumes no access to the solver source tree—only the public headers shipped with your FALCON installation, a C++ toolchain, and the input-deck format.

1.1 Syntax

UMATs are assigned per material inside the `% Materials` section using an `@UMAT:` directive. Typical forms:

```
% Materials
MyMaterial
@UMAT: /path/to/libMyModel.so Mechanical YoungsModulus=1.2e8
PoissonsRatio=0.3
%%%
```

```
% Materials
MyMaterial
@UMAT: /path/to/MyModel.cpp /path/to/MyModel.hpp Mechanical
YoungsModulus=1.2e8 PoissonsRatio=0.3
%%%
```

Notes:

- The directive token is case-insensitive and extra leading `@` characters are accepted (so `@UMAT:`, `@@umat:`, etc. are equivalent for directive lookup).
- A space after `:` is optional (`@UMAT:/path/to/lib.so ...` and `@UMAT: /path/to/lib.so ...` both work).
- Parameter passing uses `name=value` pairs; the accepted names depend on the UMAT you are using.
- For **pre-compiled libraries**, you may omit the header path entirely, or pass a placeholder like `-` / `None` (the header path is ignored in that mode).
- `CustomVariable=` can appear anywhere in the `@UMAT:` line and accepts a whitespace- and/or comma-separated list of custom variable names.

1.2 Audience & prerequisites

Topic	Expectation
Programming	Comfortable with modern C++ (C++17 or newer).
Build tools	`clang++` or `g++`, CMake or simple shell scripts — **only required if providing source code**. Users supplying pre-compiled libraries need no compiler.
Math	Familiar with stress/strain tensors in Voigt notation.
Platform	Cross-platform; tested toolchains include Linux, macOS, and Windows (via MinGW, MSVC, or WSL).

1.2.1 Deliverables and supporting files

If developing a UMAT (source code mode):

Item	Source	Action
`UMATBase.hpp`	FALCON UMAT SDK	Copy into your project (read-only; do not edit). See Appendix A for the full reference.
`StateVariable.hpp` (exported `StateVariable` enum)	FALCON UMAT SDK	Copy into your project (read-only; do not edit). See Appendix A for the full reference.
Eigen 3.4+	`https://eigen.tuxfamily.org`	Download/unpack; reference include path when compiling.
Your UMAT source	User authored	Implement both the header (`SandModel.hpp`) and implementation (`SandModel.cpp`).
Build scripts	User authored	CMakeLists.txt, build.sh, etc. (optional if letting FALCON compile at runtime).

If deploying a pre-compiled UMAT (binary mode):

Item	Source	Action
Pre-compiled library	Developer-provided	Obtain `.so` (Linux), `.dylib` (macOS), or `.dll` (Windows) from the UMAT developer.
(None of the above)	—	No SDK headers, no Eigen, no compiler needed on the target machine.

> **Tip:** The SDK headers are considered fixed files—do not modify them. Place them in an `include/` directory and point your compiler there. Only files in `src/` are user specific.

Development requirements:

- The SDK headers listed above.
- Eigen 3.4+ (header-only linear algebra library). If you do not have Eigen, download the official release and set an `EIGEN_PATH` environment variable or pass `-I/path/to/eigen` to your compiler.
- An optional scripting environment (Python/Matlab) for verification.

Deployment requirements (pre-compiled mode):

- Only the compiled shared library file.
- The library must match the target platform's architecture and ABI.

1.3 Quick start checklist

1. Set up a dedicated folder for your UMAT project.
2. Copy the FALCON UMAT SDK headers into an `include/` directory, or point your compiler to the official install location.
3. Implement the required UMAT interface in a `.cpp`/`.hpp` pair.
4. Compile both files into a shared library (`.so` on Linux, `.dylib` on macOS, `.dll` on Windows).
5. Reference the shared library from your FALCON input deck with an `@UMAT:` block.
6. Run a small verification case to confirm stability and diagnostics.

1.4 How the UMAT loader works

FALCON provides a UMAT loader that manages the loading and (optionally) compilation of UMAT libraries. Understanding how it operates helps you decide whether to supply source code or pre-compiled binaries.

1.4.1 Two modes of operation

Mode	What you provide	Compiler required?	When to use
Source compilation	`.cpp` + `.hpp` files	Yes	Development, portability across systems
Pre-compiled library	`.so`, `.dll`, `.dylib`, or versioned variants	No	Production deployments, binary distribution

1.4.2 Auto-detection of pre-compiled libraries

The UMAT loader automatically detects whether you've supplied source code or a pre-compiled library by examining the file path:

Recognised library extensions: - `.so` (Linux) - `.dll` (Windows) - `.dylib` (macOS) - `.bundle` (macOS plugin) - **Versioned libraries**: `libfoo.so.1`, `libfoo.so.1.2.3`, `libbar.1.dylib`

If a library file is detected, FALCON **skips compilation entirely** and loads the binary directly. This means:

- **No compiler is needed** on the target machine
- **No Eigen installation** is required (it was baked in at compile time)
- The library is loaded immediately using the platform's dynamic loader APIs

1.4.3 Using pre-compiled libraries in the input deck

When supplying a pre-compiled library, simply provide the library path where you would normally provide the `.cpp` path:

```
% Pre-compiled library (no compilation, no compiler needed)
SandMaterial
@UMAT:/path/to/libSandModel.so  Mechanical \
    YoungsModulus=1.2e8 PoissonsRatio=0.3

% Or with a versioned library
ClayMaterial
@UMAT:/path/to/libClayModel.so.2.1  Mechanical \
    Cohesion=25.0 FrictionAngle=28.0
```

When supplying source code, provide both `.cpp` and `.hpp` paths (FALCON compiles them):

```
% Source code (requires compiler)
SandMaterial
@UMAT:/path/to/SandModel.cpp /path/to/SandModel.hpp Mechanical \
    YoungsModulus=1.2e8 PoissonsRatio=0.3
```

Notes:

- The header path is primarily used for **rebuild detection** (so edits to the header trigger recompilation). Your `.cpp` must still `#include` whatever headers it needs.
- If you are using a **pre-compiled library**, the header path is ignored; you may omit it entirely (or pass `-` / `None` as a placeholder).

1.4.4 Smart recompilation (source mode only)

When you provide source code, the UMAT loader tracks file modification times:

1. If the compiled library **does not exist** → compile
2. If the `.cpp` or `.hpp` is **newer** than the library → recompile
3. If the library is **up-to-date** → skip compilation, load existing binary

This means repeated runs with unchanged source code are fast—no redundant compilation.

1.4.5 Cross-platform compiler handling

When compilation is needed, the UMAT loader selects an appropriate compiler:

Platform	Compiler selection order
Windows	1. `$CXX` env var, 2. MSVC `cl.exe` (if in PATH), 3. MinGW `g++`
macOS	1. `$CXX` env var, 2. Homebrew LLVM `clang++`, 3. System `clang++`
Linux	1. `$CXX` env var, 2. `clang++` (if available), 3. `g++`

Multi-token CXX is supported. You can set `CXX="ccache clang++"` or `CXX="clang++ -stdlib=libc++"` and the UMAT loader will parse it correctly.

Platform-specific compilation:

Platform	Shared library flag	OpenMP handling
Linux	`-shared`	`-fopenmp`
macOS	`-dynamiclib`	`-Xpreprocessor -fopenmp` + libomp linking with rpath
Windows (MSVC)	`/LD`	`/openmp`
Windows (MinGW)	`-shared`	`-fopenmp`

Platform	Shared library flag	OpenMP handling

1.4.6 Eigen dependency

- **Source mode**: The UMAT loader searches for Eigen in standard locations or uses `$EIGEN_PATH`
- **Pre-compiled mode**: Eigen headers were included at compile time—no runtime dependency

Eigen search paths (when compiling):

Platform	Locations checked
Linux	`/usr/include/eigen3`, `/usr/local/include/eigen3`, `/opt/eigen3`
macOS	`/opt/homebrew/include/eigen3`, `/usr/local/include/eigen3`, `/opt/local/include/eigen3`
Windows	`C:/vcpkg/installed/x64-windows/include/eigen3`, `C:/Program Files/eigen3`, `C:/eigen3`

Set `EIGEN_PATH=/your/path/to/eigen` to override.

1.4.7 Error handling

The UMAT loader provides detailed error messages:

Scenario	Error message
Source file not found	`UMAT source/library file not found: <path>`
Compilation failed	`Failed to compile UMAT model at <path> (exit code: N)`
Library won't load	`Cannot open UMAT library: <path>. Error: <dlerror message>`
Symbol not found	`Cannot load symbol '<name>': <dlerror message>`

1.4.8 Summary: source vs. pre-compiled

Aspect	Source code	Pre-compiled library
Portability	High (compiles on target)	Low (must match target ABI)
Deployment simplicity	Lower (needs compiler + Eigen)	Higher (single binary)

Aspect	Source code	Pre-compiled library
Compilation overhead	First run only (then cached)	None
Debugging	Easier (can add flags)	Harder
Recommended for	Development, sharing code	Production, binary distribution

---

1.5 Constitutive context

1.5.1 What your UMAT must do

At its core, a UMAT has **two jobs**:

1. **Update stresses and history** when the solver gives you a strain increment
2. **Provide tangent matrices** so the solver can build the global stiffness

Think of it like this: FALCON asks "If I apply this small strain increment, what stress increment do I get?" Your UMAT computes that stress change and also tells FALCON "and here's how stiff the material is right now" (the tangent matrix).

1.5.2 The math behind it (optional reading)

FALCON solves equilibrium using Newton–Raphson iterations. At every integration point (Gauss point), it needs to know how stress changes with strain:

$$\Delta\boldsymbol{\sigma}' = \mathbf{D}_{ep}\,\Delta\boldsymbol{\varepsilon} + \mathbf{S}_{ep}\,\Delta p_c + \mathbf{V}_{ep}\,\Delta\dot{\boldsymbol{\varepsilon}}$$

This equation says: "stress increment = elastic-plastic tangent × strain increment + suction coupling vector × suction change + velocity coupling × strain-rate change"

Breaking this down:

- $\mathbf{D}_{ep}$ (6×6 matrix): The **elastoplastic tangent**, also called "Dep". This relates stress increments to strain increments. Every UMAT must provide this.
- $\mathbf{S}_{ep}$ (6×1 vector): The **suction coupling vector**, also called "Sep". This relates stress increments to changes in matric suction $p_c = p_a - p_w$. Only needed for **coupled hydro-mechanical** analyses (saturated or unsaturated soils with pore pressure). For uncoupled mechanics, leave this at zero.
- $\mathbf{V}_{ep}$ (6×6 matrix): The **velocity coupling matrix**, also called "Vep". This captures rate-dependent behavior (viscoplasticity, creep). For rate-independent materials, leave this at zero.

1.5.3 The `constitutiveFlag` system

FALCON doesn't always need all three matrices. To save computation, it uses a flag to tell your UMAT what to compute:

`constitutiveFlag`	What FALCON wants	What your UMAT should fill
`0`	Standard tangent	`stressStrainMatrix` = $\mathbf{D}_{ep}$ (6×6)
`1`	Suction coupling	`sep` = $\mathbf{S}_{ep}$ (6×1 vector)
`2`	Velocity coupling	`Vep` = $\mathbf{V}_{ep}$ (6×6)

Simple example: For a basic elastoplastic model (no suction coupling, no rate effects): - When `constitutiveFlag == 0`: compute and return $\mathbf{D}_{ep}$ - When `constitutiveFlag == 1`: just set `sep[0...5] = 0.0` (already initialized to zero) - When `constitutiveFlag == 2`: just set `Vep[i][j] = 0.0` (already initialized to zero)

1.5.4 Two key UMAT functions

1. **`calculateStressIncrement`**: Called every time the solver needs a stress update
   - **Input** (all provided in `InputData`):
     - **Increments**: strain increment, velocity increment (for rate-dependent models)
     - **Pore pressure increments**: water pressure, air pressure (for coupled hydro-mechanical analysis)
     - **Saturation increment**: change in degree of saturation (for unsaturated soils)
     - **Current state**: stress, strain, void ratio, saturation, pore pressures (all in `stateVariables`)
     - **Your custom history**: plastic strains, hardening variables, etc. (in `customStateVariables`)
     - **Sensitivity derivatives**: $\partial S_w/\partial p_c$, $\partial S_w/\partial e$ (for advanced coupled analysis)
     - **Element/Gauss point info**: element number, local coordinates (for debugging/logging)
   - **Output** (must be filled in `OutputData`): stress increment, updated state variables, updated custom history
   - **Your job**: Run your constitutive law (elastic predictor, plastic corrector, etc.) and write the results
2. **`computeStressStrainMatrix`**: Called when the solver needs tangent matrices
   - **Input**: same state snapshot as above, plus `constitutiveFlag` telling you which matrix to compute
   - **Output** (fill in `OutputData`): the requested tangent matrix (Dep, Sep, or Vep)
   - **Your job**: Compute and return the matrices requested by `constitutiveFlag`

1.6 UMAT development roadmap

The FALCON UMAT workflow follows six checkpoints. Each later section in this guide expands the items below, and the flow chart highlights the key sub-tasks inside Step 3.

1. **Plan the constitutive behaviour.** Decide on stresses, strains, internal variables, and material parameters (Step 1).
2. **Lay out your project.** Copy the SDK headers, set up `include/` and `src/`, and keep user code isolated (Step 2).
3. **Implement the contract.** Derive your class from `UMATBase`, honour the runtime data exchange, and wire the six exports (Step 3).
4. **Build the shared library.** Use CMake or a shell script to produce the `.so`/`.dylib`/`.dll` that ships with your deck (Step 4).
5. **Declare the UMAT in the deck.** Reference both sources, pass every required parameter, and list custom variables (Step 5).
6. **Verify and iterate.** Smoke-test, run convergence checks, and add regression suites before deploying (Step 6 and Troubleshooting).

   Each step is detailed below.

1.7 Step 1 – Plan your constitutive model

1. **Decide on primary variables**: stress measure, strain increment handling, internal variables (e.g. plastic strains, hardening scalars).
2. **List state variables**: partition them into **standard** (provided by FALCON, see State Variables) and **custom** (your additions). Typical standard entries include:
   - `StressXX ... StressXY` (six components, tension positive)
   - `StrainXX ... StrainXY` (total strains)
   - `PoreWaterPressure`, `PoreAirPressure` (pore water and air pressures follow the "compression positive" sign convention), `VoidRatio`, `DegreeOfSaturation`
3. **Name custom variables**: choose concise identifiers, e.g. `IsotropicHardening`, `PlasticMultiplier`, `FabricTensor11`.
4. **Define material parameters**: decide which inputs the user must supply (`YoungsModulus`, `PoissonsRatio`, `Cohesion`, etc.).

Document everything in a short design note—you will use this when filling the input deck and the `getRequiredVariableName` function.

1.8 Step 2 – Structure your UMAT project

A recommended layout with starter files:

```
umat_sand/
|– include/
```

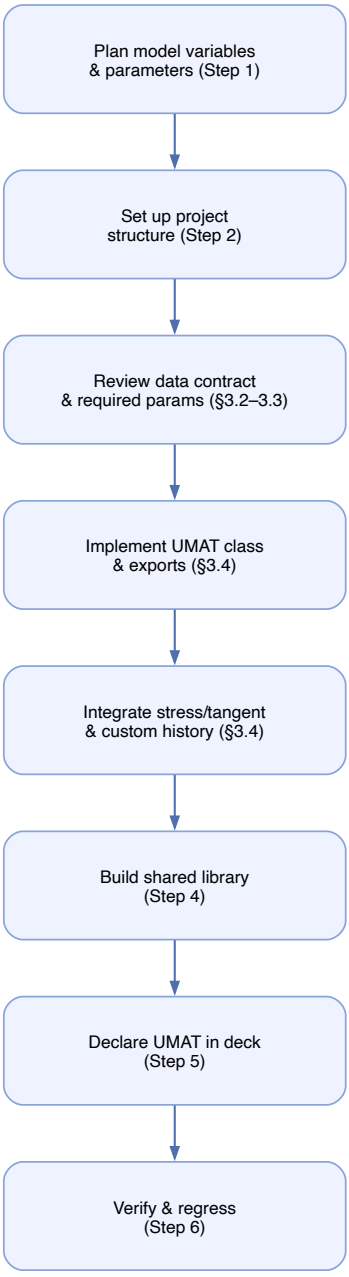

Figure 1: UMAT implementation flow chart

```
|   |– UMATBase.hpp        # from the FALCON SDK
|– src/
|   |– SandModel.hpp
|   |– SandModel.cpp
|– CMakeLists.txt          # or build.sh
|– README.md
```

Path	Description
`include/UMATBase.hpp`	Fixed header copied from the SDK (see Appendix A). Add other SDK headers here if required.
`src/SandModel.hpp`	Your UMAT class declaration. Define material parameters and helper functions.
`src/SandModel.cpp`	Implementation plus the `extern "C"` exports.
`CMakeLists.txt` or `build.sh`	Build script that compiles `SandModel.cpp` into a shared library.
`README.md`	Optional notes about assumptions, parameters, test cases.

Tip: treat `include/` as read-only. All user edits should happen in `src/`.

Important (runtime compilation): When FALCON compiles a UMAT from source, it compiles the `.cpp` file and automatically adds an Eigen include path, but it does **not** automatically add your project's `include/` directory. Make sure your UMAT source can find `UMATBase.hpp` / `StateVariable.hpp` by either:

- Including them via a relative path from your `.cpp` (for example `#include "../include/UMATBase.hpp"`), or
- Setting `CXX` to include `-I...` flags (for example `export CXX="clang++ -I/path/to/umat_project/include"`). ## State vs. custom variables

FALCON distinguishes between two histories at every integration point:

- State variables are the built-in entries managed by the solver. They cover stresses, strains, pore pressures, saturation, permeability, and related quantities listed in the StateVariables reference. UMATs receive them through `InputData` and should treat them as read-only snapshots.
- Custom variables are user-defined names that you append to `CustomVariable=` in the `@UMAT` declaration (see the materials section for syntax).
  - **Purpose.** They capture model-specific memory such as plastic hardening scalars, fabric tensors, integration diagnostics, or anything else that is not part of the built-in state inventory.

- **Storage.** FALCON creates an entry for every name you provide and keeps it in a per-Gauss-point map within `InputData`/`OutputData`. Each call to `calculateStressIncrement` should write updated values to `OutputData::updatedCustomStateVariables`; otherwise the solver will reuse the old values.
- **Lifecycle.** Unlike state variables, custom variables are not updated automatically. Your UMAT is responsible for initialising them via `initializeCustomVariable`, evolving them during stress integration, and writing any diagnostics you want to track.
- **Naming.** Choose concise, descriptive names and keep them stable across versions so old decks remain compatible.

In short, state variables reflect the common FALCON inventory, while custom variables capture anything unique to your constitutive model and must be managed by your UMAT.

1.9 Step 3 – Implement the UMAT API

1.9.1 Implementation quick reference

Task	Where	Notes
Define your UMAT class	`src/<Model>.hpp`	Inherit from `UMATBase`, store material constants, expose helper methods.
Integrate stress & update state	`src/<Model>.cpp::calculateStressIncrement`	Read from `InputData`, never mutate it; populate stress increments and history.
Return the tangent(s)	`src/<Model>.cpp::computeStressStrainMatrix`	Obey `constitutiveFlag` so the solver only receives the matrix it asked for.
Parse material parameters	`initializeUMATProperties` (inside `extern "C"`)	Build a name→value map, validate, and construct your concrete UMAT instance.
Seed custom history	`initializeCustomVariable`	Set starting values for every custom variable advertised in the deck.

1.9.2 Base class

Create a concrete class that derives from `UMATBase` and stores every material constant you collect during initialisation. At minimum the class must:

- Accept a name→value map in the constructor and cache the parameters your model needs.
- Override `calculateStressIncrement` to perform the stress integration.

- Override `computeStressStrainMatrix` to return the requested tangent (`stressStrainMatrix`, `sep`, or `Vep` depending on `constitutiveFlag`).

Optional helper methods (elastic predictor, plastic corrector, hardening updates, etc.) keep the overrides focused on orchestrating the workflow rather than the detailed algebra.

1.9.3 Runtime data contract

FALCON treats each UMAT call as a pure function of the state passed in. `InputData` arrives fully populated by the solver, while `OutputData` starts zeroed and must be filled by your code.

Solver → UMAT: `InputData`

Field	Type	Access	Role
`strainIncrement[6]`	`double[6]`	Read-only	Total strain increment in Voigt order (`xx, yy, zz, zy, zx, xy`).
`velocityIncrement[6]`	`double[6]`	Read-only	Increment of the symmetric velocity gradient; zero in quasi-static runs.
`poreWaterPressureIncrement`	`double`	Read-only	Water pressure increment for coupled hydro-mechanical cases.
`poreAirPressureIncrement`	`double`	Read-only	Air pressure increment (zero if not tracked).
`saturationIncrement`	`double`	Read-only	Change in degree of saturation.
`constitutiveFlag`	`int`	Read-only	0 = return `Dep`, 1 = fill `S_ep`, 2 = fill `V_ep` (see Constitutive context).

Field	Type	Access	Role
`initialization MethodFlag`	`int`	Read-only in `calculateStress Increment` / `computeStress StrainMatrix`	In `initialize CustomVariable` (which receives `InputData&`), your UMAT may set this to request solver-side initialisation (for example, fully-coupled unsaturated equilibration).
`dSw_dpc` / `dSw_de`	`double`	Read-only	Sensitivity of saturation to suction / void ratio; optional, defaults to zero.
`stateVariables`	`std:: vector<double>`	Read-only	Built-in state vector indexed by `State Variable`.
`customState Variables`	`std::unordered_ map<std::string, double>`	Read-only	Previous values of your custom history terms.
`elementNumber`	`int`	Read-only	Owning finite element, handy for logging.
`gaussCoords`	`std:: vector<double>`	Read-only	Local Gauss point coordinates (e.g. `[xi, eta, zeta]`).

> **Tip:** Guard custom entries defensively. If your model requires a custom variable, check it exists (and has a valid value) and throw a clear error early if it is missing.

UMAT → Solver: `OutputData`

Field	Type	Access	Role
`stress Increment[6]`	`double[6]`	Write	Effective stress increment to accumulate into the global stress vector.

Field	Type	Access	Role
`stressStrainMatrix[6][6]`	`double`	Write	Consistent tangent `Dep`; only populate if `constitutiveFlag == 0`.
`sep[6]`	`double[6]`	Write	Suction coupling **vector** (6×1); required when `constitutiveFlag == 1`; leave zero otherwise.
`Vep[6][6]`	`double`	Write	Velocity coupling tensor, required when `constitutiveFlag == 2`; leave zero otherwise.
`updatedStateVariables`	`std::vector<double>`	Write	Present in the UMAT ABI for compatibility, but **currently not consumed by the solver**. Do not rely on this to update state variables during analysis.
`updatedCustomStateVariables`	`std::unordered_map<std::string, double>`	Write	New values for every custom variable your UMAT manages.

All matrices and arrays arrive initialised to zero. A typical update looks like:

```
// Voigt order used by FALCON: [xx, yy, zz, zy, zx, xy]
constexpr int XX = 0;
constexpr int YY = 1;
constexpr int ZZ = 2;
constexpr int ZY = 3;
constexpr int ZX = 4;
constexpr int XY = 5;

// Solver → UMAT increments (read-only)
const double dEpsXX = in.strainIncrement[XX];
```

```
const double dEpsYY = in.strainIncrement[YY];
const double dEpsZZ = in.strainIncrement[ZZ];
const double dEpsZY = in.strainIncrement[ZY];
const double dEpsZX = in.strainIncrement[ZX];
const double dEpsXY = in.strainIncrement[XY];

const double dVelXX = in.velocityIncrement[XX];
const double dVelYY = in.velocityIncrement[YY];
const double dVelZZ = in.velocityIncrement[ZZ];
const double dVelZY = in.velocityIncrement[ZY];
const double dVelZX = in.velocityIncrement[ZX];
const double dVelXY = in.velocityIncrement[XY];

// Coupled analysis increments (zero in uncoupled runs)
const double dPw = in.poreWaterPressureIncrement;
const double dPa = in.poreAirPressureIncrement;
const double dSw = in.saturationIncrement;

// Current state snapshot (read-only)
const double sigmaXX = in.stateVariables[StateVariable::StressXX];
const double sigmaYY = in.stateVariables[StateVariable::StressYY];
const double sigmaZZ = in.stateVariables[StateVariable::StressZZ];
const double sigmaZY = in.stateVariables[StateVariable::StressZY];
const double sigmaZX = in.stateVariables[StateVariable::StressZX];
const double sigmaXY = in.stateVariables[StateVariable::StressXY];

const double voidRatio = in.stateVariables[StateVariable::VoidRatio];
const double saturation =
in.stateVariables[StateVariable::DegreeOfSaturation];

// Optional custom history: guard access so missing keys fail clearly.
double isoHardening = 0.0;
if (auto it = in.customStateVariables.find("IsotropicHardening"); it !=
in.customStateVariables.end()) {
    isoHardening = it->second;
} else {
    throw std::runtime_error("Missing custom variable 'IsotropicHardening'
    (declare it via CustomVariable=...)");
}

// Run your constitutive law here (elastic predictor, plastic corrector,
etc.)
// - Compute stress increment Δσ' (Voigt) and update your custom history.
double dSigma[6] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
```

```
// ... your model fills dSigma (and updates any internal/history variables
it owns) ...

// UMAT → Solver outputs
out.stressIncrement[XX] = dSigma[XX];
out.stressIncrement[YY] = dSigma[YY];
out.stressIncrement[ZZ] = dSigma[ZZ];
out.stressIncrement[ZY] = dSigma[ZY];
out.stressIncrement[ZX] = dSigma[ZX];
out.stressIncrement[XY] = dSigma[XY];

// Update custom history (your UMAT controls these values)
out.updatedCustomStateVariables["IsotropicHardening"] = isoHardening; // or
updated value

// updatedStateVariables is part of the ABI but is not currently consumed by
the solver.
// Leave it untouched (zeros) unless you have a very specific reason.
```

Tangents/couplings (`stressStrainMatrix`, `sep`, `Vep`) are returned from `computeStressStrainMatrix` (not from `calculateStressIncrement`).

Always leave fields you do not own at their zero initialisation. Do not mutate `inputData` in `calculateStressIncrement` / `computeStressStrainMatrix` (it is passed as `const`). The one exception is `initializeCustomVariable`, which receives `UMATBase::InputData&` and is explicitly intended as an in/out initialisation hook.

1.9.4 Initialisation and required parameters

Every UMAT shared library must expose **at least** these six C-style functions. FALCON calls them in a well-defined order to discover your material parameters, create an instance of your model, exchange state data every step, and keep custom variables synchronised.

Function	When FALCON calls it	What your implementation must do
`int getNumRequiredVariables()`	During deck parsing, before any analysis begins	Return the exact count of material parameters your UMAT expects. This lets the solver validate the input deck early.

Function	When FALCON calls it	What your implementation must do
`const char* getRequiredVariableName(int index)`	Immediately after the previous call, once per index `0...N-1`	Provide the canonical name for each required parameter. Names must match the tokens that appear in the deck (case-sensitive).
`void initializeUMATProperties(const char** names, const double* values, int count)`	During model setup (may be called more than once)	Translate the incoming arrays into a map (e.g. `unordered_map`), verify required parameters, apply defaults, and construct/persist the model state you need. **Be idempotent:** the solver may call this multiple times (for example once per element using the UMAT). If you need multiple parameterisations of the same UMAT code in one run, use separate libraries or implement your own instance dispatching.

Function	When FALCON calls it	What your implementation must do
`void initializeCustomVariable(UMATBase::InputData& in)`	Once per Gauss point during model initialisation (and optionally again for element activation/birth and after equilibrium "finalize")	Seed and/or **condition** your custom history in `in.customStateVariables` based on the current equilibrated snapshot in `in.stateVariables` (stress/void ratio/pore pressures, etc.). If your workflow uses a post-step equilibrium action (see `Establish Equilibrium`), this hook is called again so your UMAT can re-initialise internal variables from the established initial state. If your model needs solver-side initialisation for fully-coupled unsaturated runs, you may set `in.initializationMethodFlag = 1` to request the solver's equilibration pass (the solver will then update saturation/χ and call `initializeCustomVariable` during the Picard loop).
`void calculateStressIncrement(const UMATBase::InputData& in, UMATBase::OutputData& out)`	At every constitutive update	Read the solver snapshot, integrate your constitutive law, and populate `out.stressIncrement` plus `out.updatedCustomStateVariables`. Never mutate `in`.

Function	When FALCON calls it	What your implementation must do
`void computeStressStrainMatrix(const UMATBase::InputData& in, UMATBase::OutputData& out)`	Whenever the solver requests a tangent (usually in the same call right after `calculateStressIncrement`)	Fill the matrix corresponding to `in.constitutiveFlag`: `stressStrainMatrix` for elastic/plastic stiffness, `sep` for suction coupling, `Vep` for velocity coupling. Leave unused matrices at zero.

Minimal implementations

```
static LinearElasticUMAT* umatInstance = nullptr;

extern "C" {

int getNumRequiredVariables()
{
    return 2; // e.g. YoungsModulus, PoissonsRatio
}

const char* getRequiredVariableName(int index)
{
    static const char* names[] = { "YoungsModulus", "PoissonsRatio" };
    return (index >= 0 && index < 2) ? names[index] : nullptr;
}

void initializeUMATProperties(const char** names,
                              const double* values,
                              int count)
{
    std::unordered_map<std::string, double> props;
    for (int i = 0; i < count; ++i)
        props[names[i]] = values[i];

    if (umatInstance != nullptr) {
        delete umatInstance;
        umatInstance = nullptr;
    }
    umatInstance = new LinearElasticUMAT(props);
}
```

```
void initializeCustomVariable(UMATBase::InputData& in)
{
    (void)in; // purely elastic model—no custom history to seed
}

void calculateStressIncrement(const UMATBase::InputData& in,
                              UMATBase::OutputData& out)
{
    if (umatInstance) umatInstance->calculateStressIncrement(in, out);
}

void computeStressStrainMatrix(const UMATBase::InputData& in,
                               UMATBase::OutputData& out)
{
    if (umatInstance) umatInstance->computeStressStrainMatrix(in, out);
}

} // extern "C"
```

> **Important:** The parameter keywords in your input deck must match the strings returned by `getRequiredVariableName` exactly. For the example above, the deck must advertise `YoungsModulus=` and `PoissonsRatio=`—any spelling or casing change prevents the solver from wiring the values into your UMAT.

For the minimal elastic model above, a matching material block looks like:

```
MyElasticMaterial
@UMAT:/abs/path/umat_elastic/src/LinearElasticUMAT.cpp
/abs/path/umat_elastic/include/LinearElasticUMAT.hpp Mechanical \
    YoungsModulus=1.2e8 PoissonsRatio=0.30
```

This snippet compiles `LinearElasticUMAT.cpp`, loads the `LinearElasticUMAT` instance initialised with the two parameters, and ensures the solver provides names that align with `getRequiredVariableName`.

Function roles in practice

1. FALCON interrogates `getNumRequiredVariables` and `getRequiredVariableName` while parsing the deck so that it can assemble the `names[]`/`values[]` arrays.
2. With that list populated, it calls `initializeUMATProperties`. Treat it as an idempotent setup hook: it may be called more than once, so either (a) accept repeated identical initialisation or (b) detect parameter changes and throw a clear error.
3. For each Gauss point created in the mesh, `initializeCustomVariable` fires once. Use it to write defaults (zero hardening, flags = 0.0, etc.) so that the very first constitutive

update starts from a known state.

4. During the analysis loop, `calculateStressIncrement` and `computeStressStrainMatrix` are called repeatedly—sometimes multiple times per Newton iteration. They must be pure functions of the supplied `InputData` and the material properties you cached earlier.

Treat the initialisation hooks (`get*`, `initializeUMATProperties`, `initializeCustomVariable`) as the only place to read user-supplied material parameters. Persist them in member variables so that later `calculateStressIncrement` calls can use them cheaply.

1.9.5 Export the C interface

Define the six exported functions from the table above at the bottom of your implementation file so the solver can link against them:

```
extern "C" {
    int getNumRequiredVariables();
    const char* getRequiredVariableName(int index);
    void initializeUMATProperties(const char** variableNames,
                                  const double* variableValues,
                                  int variableCount);
    void calculateStressIncrement(const UMATBase::InputData& in,
                                  UMATBase::OutputData& out);
    void computeStressStrainMatrix(const UMATBase::InputData& in,
                                   UMATBase::OutputData& out);
    void initializeCustomVariable(UMATBase::InputData& in);
}
```

Use `initializeUMATProperties` to parse the deck parameters, route the two callback functions to your class instance, and populate `initializeCustomVariable` with default history. The declarations above mirror the runtime order documented in the previous section.

> **Tip:** keep the pointer to your concrete UMAT (for example, `static LinearElasticUMAT* umatInstance`) in an anonymous namespace so it retains state between calls without leaking the symbol.

1.10 Step 4 – Compile the shared library

> **Note:** This step is **only required if you provide source code**. If you supply a pre-compiled library (`.so`, `.dll`, `.dylib`), skip to Step 5. See How the UMAT loader works for details.

You have three options for compilation:

1. **Let FALCON compile at runtime** — provide `.cpp`/`.hpp` paths in the deck; FALCON compiles automatically

2. **Pre-compile with CMake** — build once, distribute the binary
3. **Pre-compile with a shell script** — lightweight alternative to CMake

Options 2 and 3 are shown below. For option 1, simply provide source paths in your deck and ensure a compiler is available.

1.10.1 Using CMake

```
cmake_minimum_required(VERSION 3.18)
project(SandModel LANGUAGES CXX)

add_library(SandModel SHARED src/SandModel.cpp)
target_include_directories(SandModel PRIVATE include ${EIGEN3_INCLUDE_DIR})
target_compile_features(SandModel PRIVATE cxx_std_20)
target_compile_definitions(SandModel PRIVATE EIGEN_NO_DEBUG)
```

Build & install:

```
cmake -S . -B build -DEIGEN3_INCLUDE_DIR=/path/to/eigen
cmake --build build --config Release
```

1.10.2 Using a shell script (Linux/macOS)

```
#!/usr/bin/env bash
cxx=${CXX:-clang++}
eigen=/opt/eigen-3.4.0

mkdir -p build

uname_s="$(uname -s)"
if [[ "$uname_s" == "Darwin" ]]; then
  # macOS
  "$cxx" -O3 -std=c++20 -fPIC -dynamiclib \
       -Iinclude -I"$eigen" \
       src/SandModel.cpp \
       -o build/SandModel.dylib
else
  # Linux
  "$cxx" -O3 -std=c++20 -fPIC -shared \
       -Iinclude -I"$eigen" \
       src/SandModel.cpp \
       -o build/SandModel.so
fi
```

Ensure the output is readable by FALCON and remains in a fixed location (absolute paths in the deck work best).

Library naming and reuse

- When providing **source code**, FALCON derives the shared-library name from your `.cpp` file:
  - Linux → `SandModel.cpp` → `SandModel.so`
  - macOS → `SandModel.cpp` → `SandModel.dylib`
  - Windows → `SandModel.cpp` → `SandModel.dll`
- **Smart caching**: If a compiled library already exists beside the `.cpp` file and is newer than both the source and header, FALCON skips recompilation and loads the existing binary.
- **Pre-compiled option**: You can skip runtime compilation entirely by providing a pre-built library path directly. The UMAT loader recognises `.so`, `.dll`, `.dylib`, `.bundle`, and versioned variants like `libfoo.so.1.2.3`. See How the UMAT loader works.

1.11 Step 5 – Declare the UMAT in the input deck

```
% Materials
SandMaterial
@UMAT:/home/user/umat_sand/src/SandModel.cpp
/home/user/umat_sand/src/SandModel.hpp Mechanical \
```

```
    YoungsModulus=1.2e8 PoissonsRatio=0.3 Cohesion=15.0 FrictionAngleDeg=30
\
    CustomVariable=IsotropicHardening
@PhaseChar: Solid rhos 2.65
@SWRC: vanGenuchten alpha_1 0.002 n 1.4 m 0.8 omega_prime 2.0 SW_max 1
SW_min 0
%%%
```

Guidelines:

- Use absolute paths for reliability.
- The category token (`Mechanical` here) must match the property group.
- `CustomVariable=` lists your variables (your UMAT should initialise and update them).
- Provide every parameter advertised via `getRequiredVariableName`.
- For a complete elastic walkthrough, see Example 1 – Linear elasticity below and update the deck paths as shown above.

1.12 Step 6 – Verify the UMAT

1.12.1 Smoke test

1. Create a minimal single-element problem (e.g. plane-strain, unit cube).
2. Apply a small strain increment and run FALCON.
3. Check the log for:
    - Compilation messages (“Compiled → …”, “Loaded …”).
    - Missing symbol errors (indicate incorrect exports).

1.12.2 Convergence test

- Run a multi-step loading path (cyclic shear, triaxial compression).
- Plot stress vs. strain to confirm expected trends.
- Ensure consistent tangents (`getDep`) are supplied—Newton iterations should converge quadratically near the solution.

1.12.3 Regression setup

- Save decks and parameter sets in a `tests/` folder.
- Automate by running `falcon` (or the `FALCON` binary installed on your system).
- Track critical outputs (forces, displacements, plastic strain) to catch regressions.

1.13 Troubleshooting

1.13.1 Compilation & Loading Issues

Symptom	Likely cause	Remedy
`Failed to compile UMAT` in log	Compiler not found or Eigen path wrong.	Set `CXX` or `EIGEN_PATH` environment variables; confirm file permissions.
`UMAT source/library file not found`	Path in deck is incorrect or file missing.	Verify the path exists; use absolute paths for reliability.
`Cannot open UMAT library: ... Error: ...`	Library ABI mismatch, missing dependencies, or wrong architecture.	Recompile on the target machine or check `ldd/otool -L` for missing deps.
`Cannot load symbol 'functionName'`	Missing `extern "C"` wrapper or symbol not exported.	Ensure all six required functions have `extern "C"` linkage.
Compiler error with `-Wno-nan-infinity-disabled`	Using GCC instead of Clang (flag is Clang-specific).	Set `CXX=clang++` or let the UMAT loader auto-detect.
macOS: `dyld: Library not loaded: libomp.dylib`	OpenMP runtime not found.	Install via `brew install libomp`; FALCON embeds rpath automatically when compiling.
Windows: `The specified module could not be found`	DLL dependencies missing (e.g., OpenMP runtime).	Ensure `libgomp-1.dll` or MSVC runtime is in PATH or beside the DLL.

1.13.2 Pre-compiled Library Issues

Symptom	Likely cause	Remedy
Pre-compiled `.so` treated as source	Versioned filename not recognised (e.g., `libfoo.so.1` works, but edge cases may not).	Rename to standard extension or provide full path.
Library loads but symbols fail	Library compiled with different compiler/flags than expected.	Rebuild with compatible ABI; avoid mixing MSVC and MinGW.
Wrong architecture	32-bit library on 64-bit system (or vice versa).	Recompile for the correct target architecture.

1.13.3 Runtime & Constitutive Issues

Symptom	Likely cause	Remedy
`DataFormatException: Missing required UMAT functions`	One or more exports absent.	Double-check `extern "C"` names and rebuild.
Immediate failure with diagnostic flags `-1`	Integration error inside `calculateStressIncrement`.	Add logging, ensure denominators cannot reach zero, check hardening updates.
Stagnating Newton iterations	Tangent matrix not returned or inconsistent.	Populate `out.stress` `StrainMatrix` when `constitutiveFlag == 0`.
Custom variables not updated	Missing entries in `CustomVariable=` list or wrong map key.	Verify spelling; include names exactly as used in code.
Stress results are NaN or infinite	Division by zero, uninitialized variables, or invalid material parameters.	Add bounds checking; validate `PoissonsRatio < 0.5`, etc.

1.14 Reference template

We recommend keeping a template UMAT with:

- Skeleton class inheriting `UMATBase`.
- Placeholder elastic response (easy to verify).
- Pre-filled `extern "C"` functions.
- Build scripts for all platforms you support.

Fill in the constitutive logic progressively, validating each feature (plastic flow, hardening, coupling) before moving to the next. With consistent naming and the flow above, external UMATs remain easy to maintain and share across teams.

1.15 Worked examples

The checklists above describe each phase of the UMAT workflow. The walkthroughs below show how those steps line up for concrete models.

1.15.1 Example 1 – Linear elasticity

This walkthrough follows the six-step roadmap using a Hooke's law material. Each paragraph highlights the decisions you make and points back to the relevant guidance above.

1. **Plan (Step 1).** Because the model is purely elastic, the solver only needs the six stress and strain components and no custom history. Document `YoungsModulus` and `Poissons Ratio` as material parameters and note that neither suction nor strain-rate affects the response, so only $\mathbf{D}_{ep}$ will be non-zero.

2. **Set up the project (Step 2).** Create `umat_elastic/{include,src}` and copy `UMATBase.hpp` plus `StateVariable.hpp` from the SDK. This keeps the interface stable and matches the recommended layout.
3. **Implement the contract (Step 3).** The header/implementation below derive from `UMATBase`, cache `E` and ν, and assemble the Hooke matrix once. `calculateStressIncrement` performs the stress integration (here just $\Delta\sigma' = \mathbf{D}_{ep}\,\Delta\varepsilon$), while `computeStressStrainMatrix` returns the same tensor when `constitutiveFlag == 0`. Because the model does not couple to suction or velocity, `sep` and `Vep` remain at their zero initialisation consistent with the constitutive context.
4. **Build (Step 4).** Run the CMake recipe (or shell alternative) targeting `src/LinearElasticUMAT.cpp`. This produces a platform-specific shared library; the exact filename depends on your build tool (for example CMake often prefixes `lib` on Unix-like systems).
5. **Declare the UMAT (Step 5).** In the deck, reference both the source and header paths and pass `YoungsModulus` and `PoissonsRatio`. No custom variables are listed because the code manages no additional history.
6. **Verify (Step 6).** Execute a single-element tension or shear test. Check that stresses match analytical Hooke predictions, the solver reports the UMAT matrices as zeros where expected (`sep`, `Vep`), and diagnostic flags remain zero.

Key fragment (Step 3). The stress update is the textbook Hooke tensor applied to the strain increment:

```
for (int i = 0; i < 6; ++i) {
    out.stressIncrement[i] = 0.0;
    for (int j = 0; j < 6; ++j)
        out.stressIncrement[i] += elasticityTensor[i][j] *
inputData.strainIncrement[j];
}
```

`computeStressStrainMatrix` copies the same tensor into `stressStrainMatrix` when `constitutiveFlag == 0`; `sep` and `Vep` stay zero for all calls.

Complete listing. The header and source below align with every step described earlier—copy them into `include/LinearElasticUMAT.hpp` and `src/LinearElasticUMAT.cpp` respectively and adjust as you extend the model.

Header – `include/LinearElasticUMAT.hpp`

```
#ifndef LINEAR_ELASTIC_UMAT_HPP
#define LINEAR_ELASTIC_UMAT_HPP

#include "UMATBase.hpp"
#include <unordered_map>
```

```
#include <string>

class LinearElasticUMAT : public UMATBase {
public:
    LinearElasticUMAT(const std::unordered_map<std::string, double>&
properties);

    void initializeElasticityTensor();
    void calculateStressIncrement(const InputData& inputData, OutputData&
    outputData) override;
    void computeStressStrainMatrix(const InputData& inputData, OutputData&
    outputData) override;

private:
    double E{0.0};
    double nu{0.0};
    double elasticityTensor[6][6] = {{0.0}};
};

// Required exports (used by FALCON)
extern "C" int getNumRequiredVariables();
extern "C" const char* getRequiredVariableName(int index);
extern "C" void initializeUMATProperties(const char** variableNames,
                                         const double* variableValues,
                                         int variableCount);
extern "C" void initializeCustomVariable(UMATBase::InputData& inputData);
extern "C" void calculateStressIncrement(const UMATBase::InputData&
inputData,
                                         UMATBase::OutputData& outputData);
extern "C" void computeStressStrainMatrix(const UMATBase::InputData&
inputData,
                                          UMATBase::OutputData& outputData);

// Optional convenience accessors (not used by FALCON)
extern "C" const double* getStressIncrement(const UMATBase::OutputData&
outputData);
extern "C" const double* getStressStrainMatrixRow(const
UMATBase::OutputData& outputData, int row);
extern "C" const double* getSepVector(const UMATBase::OutputData&
outputData);
extern "C" const double* getVepRow(const UMATBase::OutputData& outputData,
int row);

#endif // LINEAR_ELASTIC_UMAT_HPP
```

Implementation – `src/LinearElasticUMAT.cpp`

```
#include "LinearElasticUMAT.hpp"
#include <stdexcept>
#include <iostream>

static LinearElasticUMAT* umatInstance = nullptr;

static const char* requiredVariables[] = {
    "YoungsModulus",
    "PoissonsRatio",
};

extern "C" int getNumRequiredVariables() {
    return static_cast<int>(sizeof(requiredVariables) /
    sizeof(requiredVariables[0]));
}

extern "C" const char* getRequiredVariableName(int index) {
    if (index < 0 || index >= getNumRequiredVariables()) return nullptr;
    return requiredVariables[index];
}

extern "C" void initializeUMATProperties(const char** variableNames,
                                         const double* variableValues,
                                         int variableCount) {
    if (variableCount < getNumRequiredVariables()) {
        throw std::invalid_argument("Variable count too small for
        LinearElasticUMAT.");
    }

    std::unordered_map<std::string, double> properties;
    for (int i = 0; i < variableCount; ++i) {
        if (!variableNames || !variableNames[i]) continue;
        properties[variableNames[i]] = variableValues[i];
    }

    if (umatInstance != nullptr) {
        delete umatInstance;
        umatInstance = nullptr;
    }

    umatInstance = new LinearElasticUMAT(properties);
}

LinearElasticUMAT::LinearElasticUMAT(const std::unordered_map<std::string,
double>& properties) {
```

```
    auto youngIt = properties.find("YoungsModulus");
    auto poissonIt = properties.find("PoissonsRatio");
    if (youngIt == properties.end() || poissonIt == properties.end()) {
        throw std::invalid_argument("Properties map must contain
        'YoungsModulus' and 'PoissonsRatio'.");
    }

    E = youngIt->second;
    nu = poissonIt->second;
    initializeElasticityTensor();
}

void LinearElasticUMAT::initializeElasticityTensor() {
    const double lambda = (E * nu) / ((1.0 + nu) * (1.0 - 2.0 * nu));
    const double mu = E / (2.0 * (1.0 + nu));

    elasticityTensor[0][0] = lambda + 2 * mu;
    elasticityTensor[0][1] = lambda;
    elasticityTensor[0][2] = lambda;
    elasticityTensor[0][3] = 0.0;
    elasticityTensor[0][4] = 0.0;
    elasticityTensor[0][5] = 0.0;

    elasticityTensor[1][0] = lambda;
    elasticityTensor[1][1] = lambda + 2 * mu;
    elasticityTensor[1][2] = lambda;
    elasticityTensor[1][3] = 0.0;
    elasticityTensor[1][4] = 0.0;
    elasticityTensor[1][5] = 0.0;

    elasticityTensor[2][0] = lambda;
    elasticityTensor[2][1] = lambda;
    elasticityTensor[2][2] = lambda + 2 * mu;
    elasticityTensor[2][3] = 0.0;
    elasticityTensor[2][4] = 0.0;
    elasticityTensor[2][5] = 0.0;

    elasticityTensor[3][0] = 0.0;
    elasticityTensor[3][1] = 0.0;
    elasticityTensor[3][2] = 0.0;
    elasticityTensor[3][3] = mu;
    elasticityTensor[3][4] = 0.0;
    elasticityTensor[3][5] = 0.0;

    elasticityTensor[4][0] = 0.0;
```

```
    elasticityTensor[4][1] = 0.0;
    elasticityTensor[4][2] = 0.0;
    elasticityTensor[4][3] = 0.0;
    elasticityTensor[4][4] = mu;
    elasticityTensor[4][5] = 0.0;

    elasticityTensor[5][0] = 0.0;
    elasticityTensor[5][1] = 0.0;
    elasticityTensor[5][2] = 0.0;
    elasticityTensor[5][3] = 0.0;
    elasticityTensor[5][4] = 0.0;
    elasticityTensor[5][5] = mu;
}

void LinearElasticUMAT::calculateStressIncrement(const InputData&
inputData, OutputData& outputData) {
    for (int i = 0; i < 6; ++i) {
        outputData.stressIncrement[i] = 0.0;
        for (int j = 0; j < 6; ++j) {
            outputData.stressIncrement[i] += elasticityTensor[i][j] *
inputData.strainIncrement[j];
        }
    }
}

void LinearElasticUMAT::computeStressStrainMatrix(const InputData&
inputData, OutputData& outputData) {
    switch (inputData.constitutiveFlag) {
        case 0: // Dep
            for (int i = 0; i < 6; ++i)
                for (int j = 0; j < 6; ++j)
                    outputData.stressStrainMatrix[i][j] =
elasticityTensor[i][j];
            break;
        case 1: // Sep
            for (int i = 0; i < 6; ++i) outputData.sep[i] = 0.0;
            break;
        case 2: // Vep
            for (int i = 0; i < 6; ++i)
                for (int j = 0; j < 6; ++j)
                    outputData.Vep[i][j] = 0.0;
            break;
        default:
            throw std::invalid_argument("Unknown constitutiveFlag in
            InputData");
```

```
    }
}

extern "C" void calculateStressIncrement(const UMATBase::InputData&
inputData,
                                         UMATBase::OutputData& outputData) {
    if (umatInstance) umatInstance->calculateStressIncrement(inputData,
    outputData);
    else std::cerr << "Error: UMAT instance not initialized.\n";
}

extern "C" void computeStressStrainMatrix(const UMATBase::InputData&
inputData,
                                          UMATBase::OutputData& outputData)
{
    if (umatInstance) umatInstance->computeStressStrainMatrix(inputData,
    outputData);
    else std::cerr << "Error: UMAT instance not initialized.\n";
}

extern "C" const double* getStressIncrement(const UMATBase::OutputData&
outputData) {
    return outputData.stressIncrement;
}

extern "C" const double* getStressStrainMatrixRow(const
UMATBase::OutputData& outputData, int row) {
    return outputData.stressStrainMatrix[row];
}

extern "C" const double* getSepVector(const UMATBase::OutputData&
outputData) {
    return outputData.sep;
}

extern "C" const double* getVepRow(const UMATBase::OutputData& outputData,
int row) {
    return outputData.Vep[row];
}

extern "C" void initializeCustomVariable(UMATBase::InputData& inputData) {
    (void)inputData;
}
```

Sample input deck excerpt

```
% Materials
LinearElasticSoil
@UMAT:/abs/path/umat_elastic/src/LinearElasticUMAT.cpp
/abs/path/umat_elastic/include/LinearElasticUMAT.hpp Mechanical \
    YoungsModulus=1.2e8 PoissonsRatio=0.30
@PhaseChar: Solid rhos 2.65
```

This mirrors the instructions in Step 5: absolute paths are recommended, only the two required parameters are supplied, and no custom variables are needed.

1.16 Appendix A – Reference SDK headers

The authoritative SDK headers are those shipped with your FALCON installation. The excerpts below reflect the current UMAT contract and are included for quick reference; prefer copying the actual header files from the SDK rather than copy-pasting from this page.

1.16.1 UMATBase.hpp

```cpp
#ifndef UMAT_BASE_HPP
#define UMAT_BASE_HPP

#include <vector>
#include <unordered_map>
#include <string>
#include "StateVariable.hpp" // for StateVariable enum

class UMATBase {
public:
    virtual ~UMATBase() = default;

    struct InputData {
        double strainIncrement[6];
        double velocityIncrement[6];
        double poreWaterPressureIncrement;
        double poreAirPressureIncrement;
        double saturationIncrement;
        /* NEW – 0: Dep  |  1: S_ep  |  2: V_ep */
        int constitutiveFlag;

        int initializationMethodFlag;
        /* ∂S_p/∂p_c  and  ∂S_p/∂e          (p_c = suction, e = void ratio)
             * They default to zero, so existing codes that never set them
             * continue to behave exactly as before.
```

```
         */
        double dSw_dpc;   // [-] per unit cap-pressure
        double dSw_de;    // [-] per unit void-ratio
        // Fixed-size vector for standard state variables
        std::vector<double> stateVariables;
        // Flexible container for custom user-defined state variables
        std::unordered_map<std::string, double> customStateVariables;
        // New fields for element number and Gauss coordinates
        int elementNumber;                // Element number
        std::vector<double> gaussCoords;  // Gauss point coordinates (e.g.,
        [x, y, z])

        InputData()
            : strainIncrement{0.0, 0.0, 0.0, 0.0, 0.0, 0.0}
            , velocityIncrement{0.0, 0.0, 0.0, 0.0, 0.0, 0.0}
            , poreWaterPressureIncrement(0.0)
            , poreAirPressureIncrement(0.0)
            , saturationIncrement(0.0)
            , constitutiveFlag(0)
            , initializationMethodFlag(0)
            , dSw_dpc(0.0)
            , dSw_de(0.0)
            , stateVariables(StateVariable::NumVariables, 0.0)
            , elementNumber(0)
        {}
    };

    struct OutputData {
        double stressIncrement[6];
        double stressStrainMatrix[6][6];
        /* S_ep vector (dσ = ... + S_ep · d p_c) returned as 6×1 */
        double sep[6];
        double Vep[6][6];
        // Fixed-size vector for standard state variables
        std::vector<double> updatedStateVariables;
        // Flexible container for custom user-defined state variables
        std::unordered_map<std::string, double> updatedCustomStateVariables;

        OutputData()
            : stressIncrement{0.0, 0.0, 0.0, 0.0, 0.0, 0.0}
            , stressStrainMatrix{{0.0}}  // Zero-initialize entire 6x6 array
            , sep{0.0, 0.0, 0.0, 0.0, 0.0, 0.0}
            , Vep{{0.0}}  // Zero-initialize entire 6x6 array
            , updatedStateVariables(StateVariable::NumVariables, 0.0)
        {}
```

```
    };

    virtual void calculateStressIncrement(const InputData& inputData,
    OutputData& outputData) = 0;
    virtual void computeStressStrainMatrix(const InputData& inputData,
    OutputData& outputData) = 0;
};

#endif // UMAT_BASE_HPP
```

1.16.2 StateVariable.hpp

```
// StateVariable.hpp
// Minimal shared StateVariable enum used by UMAT interface and core code.
#ifndef STATE_VARIABLE_HPP
#define STATE_VARIABLE_HPP

// IMPORTANT:
// Keep ordering stable. This enum is used for indexing state variable
vectors.
enum StateVariable {
    StressXX,
    StressYY,
    StressZZ,
    StressZY,
    StressZX,
    StressXY,
    StrainXX,
    StrainYY,
    StrainZZ,
    StrainZY,
    StrainZX,
    StrainXY,
    DStrainXX,
    DStrainYY,
    DStrainZZ,
    DStrainZY,
    DStrainZX,
    DStrainXY,
    VelStrainXX,
    VelStrainYY,
    VelStrainZZ,
    VelStrainZY,
    VelStrainZX,
```

```
    VelStrainXY,
    PoreWaterPressure,
    PoreAirPressure,
    InitialPoreWaterPressure,
    InitialPoreAirPressure,
    VoidRatio,
    DegreeOfSaturation,
    PcAlpha,
    SlopeR,
    SuctionOld,
    VoidRatioOld,
    DeltaVoidRaioOld,
    DeltaSuctionOld,
    PermW_XX,
    PermW_YY,
    PermW_ZZ,
    PermW_ZY,
    PermW_ZX,
    PermW_XY,
    PermA_XX,
    PermA_YY,
    PermA_ZZ,
    PermA_ZY,
    PermA_ZX,
    PermA_XY,
    Xi,
    Damping,
    alpha_p_c,
    delta_satOld,
    Saturation_old,
    TotalStressXX,
    TotalStressYY,
    TotalStressZZ,
    InitialVoidRatio,
    // ------------------------------------------------------------------
    // PML (Perfectly Matched Layer) coefficients and memory variables
    // NOTE: Appended to preserve existing indices.
    // ------------------------------------------------------------------
    PMLSigmaX,
    PMLKappaX,
    PMLAlphaX,
    PMLSigmaY,
    PMLKappaY,
    PMLAlphaY,
    PMLPsiUX_X,
```

```
    PMLPsiUY_X,
    PMLPsiUX_Y,
    PMLPsiUY_Y,
    // 3D extensions (appended to preserve existing indices)
    PMLSigmaZ,
    PMLKappaZ,
    PMLAlphaZ,
    PMLPsiUZ_X,
    PMLPsiUZ_Y,
    PMLPsiUX_Z,
    PMLPsiUY_Z,
    PMLPsiUZ_Z,
    // Coupled/Fully-coupled extensions: CPML memory variables for pore
    pressures
    PMLPsiPW_X,
    PMLPsiPW_Y,
    PMLPsiPW_Z,
    PMLPsiPA_X,
    PMLPsiPA_Y,
    PMLPsiPA_Z,
    NumVariables
};

#endif // STATE_VARIABLE_HPP
```