**Coupling a MOOSE Material with the SCIANTIX Fission Gas Release Module**

**Overview and Design**

**SCIANTIX Background:** *SCIANTIX* is an open-source, zero-dimensional code designed to model inert fission gas behavior (xenon, krypton, etc.) in UO₂ nuclear fuel at the grain scale . It uses physics-based kinetic models to compute key engineering quantities, notably fission gas release (FGR) and gaseous swelling of the fuel . The code is object-oriented and intended to be embedded into larger fuel performance codes as a module for fission gas behavior . In coupling scenarios, an external driver (here, a MOOSE-based application) provides local conditions (temperature, irradiation, etc.) to SCIANTIX, which then updates its internal state and returns quantities like intra-granular and inter-granular gas release fractions and swelling .

**MOOSE Material Coupling Approach:** In a MOOSE-based fuel performance simulation (e.g., BISON), a *Material* class is the appropriate place to integrate SCIANTIX. The Material system in MOOSE allows computing material properties at quadrature points and making them available to physics kernels (e.g. thermal conduction, mechanics). Our goal is to create a SciantixMaterial that:

* Links against the external SCIANTIX library and calls its API directly to perform fission gas release and swelling calculations.
* Accepts input parameters from the simulation: local **temperature**, **burnup** (or time-integrated exposure), and **fission rate** (or power density) — these can be provided as coupled MOOSE variables or as user-specified constants.
* Maintains SCIANTIX internal state for each material point across time steps, so that the evolution of gas release and swelling is history-dependent.
* Outputs material properties including:
  + *Intra-granular gas release fraction* (fraction of produced gas that has diffused out of the grain interior to the grain boundary),
  + *Inter-granular gas release fraction* (fraction of gas at grain boundaries released to the fuel exterior),
  + *Total fission gas release* (cumulative fraction of gas produced that has been released from the fuel to the plenum),
  + *Fuel porosity* (fraction of fuel volume occupied by gas pores, which can increase as gas bubbles form and coalesce),
  + *Swelling strain tensor* (the volumetric expansion strain caused by gas bubble swelling).

These outputs can then be used in other parts of the simulation. For example, the swelling strain tensor can act as an **eigenstrain** (pre-strain) in the mechanics module to induce fuel expansion, and the porosity can affect thermal or mechanical properties. The intra- and inter-granular release fractions might be output for diagnostics or used to compute gas pressure in the fuel/cladding gap.

**Coupling Algorithm:** Each time step (or nonlinear iteration), the Material will invoke SCIANTIX for each quadrature point (representing a material point in the fuel) to update fission gas behavior. The process follows the general strategy used in other code couplings :

1. **Data Transfer to SCIANTIX:** MOOSE provides the current local temperature, burnup/fission density, and possibly local stress state (hydrostatic pressure) to SCIANTIX. (Stress can influence gas bubble behavior; if needed, it can be passed as well, although we’ll focus on temperature and irradiation inputs here.)
2. **SCIANTIX Update:** Using the previous state of the material point (e.g. gas already released, bubble sizes, etc.) as initial conditions, SCIANTIX computes the new state over the current time step. This involves solving SCIANTIX’s internal set of ODE/PDE models for gas diffusion and bubble evolution over the time increment.
3. **Results Retrieval:** The updated values are retrieved from SCIANTIX – e.g., the new intra-granular and inter-granular release fractions and the increments in swelling or porosity.
4. **Material Property Update:** The Material class sets its output properties (porosity, swelling strain, etc.) based on SCIANTIX results for use by other MOOSE systems in the same time step.
5. **State Storage:** The new SCIANTIX state becomes the “previous” state for the next time step. (In MOOSE, we can use stateful material properties to store history, or maintain persistent SCIANTIX objects that carry state between calls.)

By repeating this each time step, the fuel performance simulation and SCIANTIX are **tightly coupled**: the fuel temperature and burnup from the MOOSE simulation drive the gas release calculation, and the resulting swelling or gas release feedback can influence the simulation (e.g., swelling-induced stress, reduced thermal conductivity due to porosity, or released gas affecting rod internal pressure if modeled elsewhere).

**SciantixMaterial**

**Class Implementation**

Below we present example C++ code for the MOOSE Material class that couples to SCIANTIX. This includes a header file (SciantixMaterial.h) and source file (SciantixMaterial.C). The code is written against the latest MOOSE APIs for Materials (compatible with the current MOOSE version in 2025), and uses the SCIANTIX library’s interface (we assume SCIANTIX provides a C++ API, which we call through a Sciantix::Simulation class in this example).

In the code, comments are provided to explain each section. (Note: In practice, you would need to adjust function names and data structures to match the actual SCIANTIX API. The example assumes a plausible interface based on SCIANTIX documentation.)

**SciantixMaterial.h**

#pragma once

// Include MOOSE material base class and required interfaces

#include "Material.h"

#include "MaterialProperty.h"

#include "RankTwoTensor.h"

// Include SCIANTIX headers (assuming Sciantix provides a Simulation class in its API)

#include "Sciantix/Simulation.h" // (Path depends on Sciantix installation)

// Forward declarations (if needed) could go here

// namespace Sciantix { class Simulation; }

/// Material class that couples with SCIANTIX to compute fission gas release and swelling in UO2 fuel.

class SciantixMaterial : public Material

{

public:

// Required constructor for MOOSE objects

SciantixMaterial(const InputParameters & parameters);

// Define valid input parameters for this class

static InputParameters validParams();

protected:

/// MOOSE override: compute properties at each quadrature point

virtual void computeQpProperties() override;

private:

// \*\* Coupled variables from MOOSE \*\*

const VariableValue & \_temperature; ///< Fuel temperature [K] at the current point

const VariableValue & \_fission\_rate; ///< Fission rate [fissions/m^3/s] at the current point

// Burnup can be taken as a coupled variable or a user parameter. Here we use a parameter for initial burnup.

const Real \_initial\_burnup; ///< Initial burnup or burnup at start of simulation [e.g., MWd/kgU]

// \*\* Output material properties \*\*

MaterialProperty<Real> & \_porosity; ///< Current porosity (fraction) of the fuel

MaterialProperty<RankTwoTensor> & \_swelling\_strain; ///< Swelling eigenstrain tensor (volumetric strain from gas swelling)

MaterialProperty<Real> & \_intragr\_release; ///< Fraction of fission gas released from within grains (intra-granular)

MaterialProperty<Real> & \_intergr\_release; ///< Fraction of fission gas released from grain boundaries to outside (inter-granular)

MaterialProperty<Real> & \_total\_fgr; ///< Total fission gas release fraction (cumulative out-of-fuel release)

// \*\* Internal state: SCIANTIX simulation objects per point \*\*

std::vector<Sciantix::Simulation> \_sims; ///< One Sciantix simulation instance per quadrature point

bool \_initialized; ///< Flag to initialize Sciantix states once

};

**Key Points in Header:** We derive from MOOSE’s Material base. We declare references to coupled variables for temperature and fission rate (these will be linked to MOOSE variables in the input file). We also have an \_initial\_burnup as a parameter (if the simulation starts at some burnup > 0 or for setting initial conditions; for fresh fuel this would be 0). The Material properties to output are declared using MaterialProperty<T> & for each quantity of interest. Finally, we plan to maintain a Sciantix::Simulation object for each quadrature point in the mesh (encapsulating the SCIANTIX state for that material point).

**Note:** In practice, managing a separate SCIANTIX simulation per integration point is necessary if conditions vary spatially. This can be memory-intensive, but SCIANTIX is designed to be efficient for 0D calculations . An alternative could be to share a single SCIANTIX instance if one assumes a homogeneous fuel cell, but here we keep them per point to allow spatial variations in temperature or burnup.

**SciantixMaterial.C**

#include "SciantixMaterial.h"

// MOOSE macro to register the object (if needed in app)

// registerMooseObject("YourAppName", SciantixMaterial);

// Define valid input parameters for this Material

InputParameters

SciantixMaterial::validParams()

{

InputParameters params = Material::validParams();

// Couple required field variables from the simulation

params.addRequiredCoupledVar("temperature", "Fuel temperature variable in the simulation");

params.addRequiredCoupledVar("fission\_rate", "Local fission rate (or power) variable in the simulation");

// Optional/constant parameters

params.addParam<Real>("burnup", 0.0, "Initial fuel burnup at simulation start (e.g., in MWd/kgU or FIMA)");

params.addParam<std::string>("sciantix\_input\_file", "",

"Path to a SCIANTIX input settings file (optional, used to configure Sciantix models)");

return params;

}

// Constructor: initialize references and material properties

SciantixMaterial::SciantixMaterial(const InputParameters & params)

: Material(params),

// Get coupled field variables

\_temperature(coupledValue("temperature")),

\_fission\_rate(coupledValue("fission\_rate")),

// Get user parameters

\_initial\_burnup(getParam<Real>("burnup")),

// Declare material properties that this material will compute

\_porosity(declareProperty<Real>("porosity")),

\_swelling\_strain(declareProperty<RankTwoTensor>("swelling\_strain")),

\_intragr\_release(declareProperty<Real>("intragr\_release")),

\_intergr\_release(declareProperty<Real>("intergr\_release")),

\_total\_fgr(declareProperty<Real>("total\_fgr")),

\_initialized(false)

{

// Reserve space for Sciantix simulations for each quadrature point in an element.

// We use Moose property to get number of points in current element's QRule:

unsigned int n\_qpoints = \_qrule->nPoints();

\_sims.resize(n\_qpoints);

// \*\*Initialize SCIANTIX simulations for each point\*\* (at t=0):

std::string sci\_input = getParam<std::string>("sciantix\_input\_file");

for (unsigned int qp = 0; qp < n\_qpoints; ++qp)

{

// Initialize each Sciantix simulation instance

if (!sci\_input.empty()) {

// If an input file is provided, let Sciantix read its configuration (models, parameters).

\_sims[qp].readInputFile(sci\_input);

}

// Set initial burnup or initial state in Sciantix

\_sims[qp].setBurnup(\_initial\_burnup);

// We could also set initial temperature, etc., but Sciantix likely uses it during stepping.

// Initialize internal variables (e.g., zero gas released, initial bubble sizes, etc.)

\_sims[qp].initialize();

}

\_initialized = true;

}

// Compute material properties at each quadrature point

void

SciantixMaterial::computeQpProperties()

{

// Ensure Sciantix is initialized (particularly after mesh adaptivity or if n\_qpoints changed)

if (!\_initialized)

{

\_sims.resize(\_qrule->nPoints());

// (Re)initialize simulations if needed, similar to constructor setup...

for (unsigned int qp = 0; qp < \_sims.size(); ++qp)

{

\_sims[qp].setBurnup(\_initial\_burnup);

\_sims[qp].initialize();

}

\_initialized = true;

}

// Get current conditions at this quadrature point

Real T = \_temperature[\_qp]; // current temperature [K] at this point

Real F = \_fission\_rate[\_qp]; // current fission rate density [fissions/m^3/s]

Real dt = this->dt(); // current time step size [s]

// Access the Sciantix simulation object for this material point (\_qp is the current point index)

Sciantix::Simulation & sim = \_sims[\_qp];

// Provide current state to SCIANTIX

sim.setTemperature(T);

sim.setFissionRate(F);

// If hydrostatic stress is needed for intergranular bubble evolution, we could do:

// Real sigma\_h = ... (e.g., from a mechanical material or auxiliary field)

// sim.setHydrostaticStress(sigma\_h);

// Advance Sciantix simulation by dt to update fission gas and swelling

sim.advance(dt); // (Assume this performs the time-integration of SCIANTIX models over dt)

// Retrieve updated results from Sciantix

Real intrag\_frac = sim.getIntraGranularReleaseFraction(); // fraction of gas produced that left grain interiors

Real interg\_frac = sim.getInterGranularReleaseFraction(); // fraction of gas that escaped from grain boundaries

Real total\_frac = sim.getTotalReleaseFraction(); // total fraction of gas released from fuel (to plenum)

Real swell\_vol = sim.getTotalSwellingFraction(); // total volumetric swelling strain (ΔV/V)

Real porosity = sim.getPorosityFraction(); // current porosity of fuel

// Assign to material properties for this quadrature point

\_intragr\_release[\_qp] = intrag\_frac;

\_intergr\_release[\_qp] = interg\_frac;

\_total\_fgr[\_qp] = total\_frac;

\_porosity[\_qp] = porosity;

// Compute the swelling strain tensor from volumetric swelling.

// We assume isotropic swelling, so the eigenstrain = (ΔV/V)/3 on each normal direction.

Real volumetric\_strain = swell\_vol;

RankTwoTensor iso;

iso.setDiagonal(volumetric\_strain / 3.0); // small-strain assumption: equally distributed in x, y, z

\_swelling\_strain[\_qp] = iso;

// (Now these properties are available to other MOOSE systems, e.g., solid mechanics or outputs.)

}

**Explanation:** In computeQpProperties(), which MOOSE calls for each integration point \_qp, we do the following:

* Ensure that the \_sims vector of Sciantix simulation objects is initialized and sized correctly (this may be needed if the number of quadrature points changes, e.g., after mesh refinement).
* Retrieve the current temperature T and fission rate F at this point from the coupled variables. We also obtain the time step size dt via this->dt() (a MOOSE API call that provides the simulation’s current timestep).
* We get a reference to the Sciantix simulation object corresponding to this quadrature point.
* **Data to Sciantix:** We call sim.setTemperature(T) and sim.setFissionRate(F) to feed the current local conditions into SCIANTIX. (If needed, other parameters like hydrostatic fuel stress could be set similarly, as commented, since grain-boundary gas release can depend on pressure.)
* **Advance Sciantix:** We call sim.advance(dt). This is a placeholder for whatever routine SCIANTIX provides to advance its solution by a time increment. Under the hood, this call will solve the Sciantix models over the interval dt given the current state and inputs. After it returns, the Sciantix object sim holds updated state (gas concentrations, bubble sizes, etc.).
* **Retrieve results:** We then query sim for the outputs we need:
  + getIntraGranularReleaseFraction() – for intra-granular release. This could represent the fraction of total fission gas atoms that have been released from the grain interior into grain boundary bubbles.
  + getInterGranularReleaseFraction() – fraction of total gas that has been released from grain boundaries to the free volume (plenum). This is effectively the fraction of gas that has left the fuel.
  + getTotalReleaseFraction() – total fission gas release fraction to the plenum. (Depending on definitions, this might be identical to the inter-granular release at end of life, but we include it for clarity. It might also include any gas released by other mechanisms.)
  + getTotalSwellingFraction() – the total volumetric swelling strain $(\Delta V/V)$ due to fission gas (sum of intra-granular bubble swelling inside grains and inter-granular bubble/pore expansion at grain boundaries).
  + getPorosityFraction() – the current porosity of the fuel. As fission gas precipitates into bubbles, especially at grain boundaries (inter-connected bubbles), the fuel develops porosity which can be computed by Sciantix. (Initially, the porosity might be the as-fabricated porosity; as gas escapes to grain boundaries and forms bubbles or tunnels, porosity increases.)
* We then assign these values to the MOOSE material properties for the current quadrature point. This makes them available for output or for use by other materials/kernels. For example, \_porosity could be used by a thermal conductivity material model to reduce thermal conductivity, \_total\_fgr could be output to track fission gas release, etc.
* Finally, we construct a **swelling strain tensor**. Assuming isotropic swelling, we distribute the volumetric swelling fraction equally on the three normal strains. We create an isotropic RankTwoTensor with diagonal components = (ΔV/V)/3 and assign it to \_swelling\_strain. This tensor can be interpreted by the solid mechanics module as an eigenstrain (thermal strain-like effect) that causes expansion. In small-strain formulation, an isotropic volumetric strain of $\epsilon\_v$ is represented by $\epsilon\_{xx}=\epsilon\_{yy}=\epsilon\_{zz}=\epsilon\_v/3$ (so that the trace $\epsilon\_{xx}+\epsilon\_{yy}+\epsilon\_{zz} = \epsilon\_v = \Delta V/V$).

**Note on Statefulness:** The Sciantix::Simulation object carries the state of the fission gas and bubbles for each point (e.g., how much gas is already released, how many gas atoms are in bubbles, bubble sizes, etc.). This state persists in our \_sims vector between calls to computeQpProperties() (the vector is a member of the Material class). Thus, when sim.advance(dt) is called, it uses the state from the previous time step as the initial condition. This is how the history effects (time accumulation of gas production and release) are accounted for. After advancing, the state is updated and stored for the next time step. This design mirrors how other codes couple with SCIANTIX: passing previous step values and current conditions to update the state each iteration .

**Thread Safety:** By default, MOOSE may evaluate materials in threads, but since each \_sims[qp] is unique to a quadrature point, and each thread works on different QPs, this design should be thread-safe (no two threads use the same Sciantix::Simulation object simultaneously). If needed, one could further ensure thread-safety by marking the material as thread\_local=false if SCIANTIX had any internal global state, but assuming it’s purely per-object, we are fine.

**Example MOOSE Input Snippet**

To use the SciantixMaterial in a MOOSE-based input file (.i file), you need to do the following:

* Define the relevant variables (temperature, burnup or fission rate, etc.) in the input file.
* Add the material block, specifying the type as our new class and linking the coupled variables.

Below is an example snippet of a MOOSE input file demonstrating how to set up the material and its inputs. This assumes you have a temperature field (solved by heat conduction or provided by an external source), and an auxvariable or userobject providing fission rate or power:

[Variables]

[./temp] # Temperature field in the fuel

initial\_condition = 300.0 # K, initial guess

family = LAGRANGE

order = FIRST

[../]

[./burnup] # (Optional) Burnup variable if burnup is solved or tracked via an AuxKernel

initial\_condition = 0.0

family = MONOMIAL

order = CONSTANT

[../]

[./fission\_rate] # Fission rate (could be computed from power)

initial\_condition = 0.0

family = MONOMIAL

order = CONSTANT

[../]

[]

[Materials]

[./fuel\_sciantix]

type = SciantixMaterial

temperature = temp # Couple the temperature field

fission\_rate = fission\_rate # Couple the fission rate (or power density field)

burnup = 0.0 # Initial burnup (MWd/kgU) at start; can be updated via burnup variable or user object

sciantix\_input\_file = "sciantix\_input.txt" # Path to a Sciantix input settings file with model parameters (optional)

[../]

[]

[Outputs]

exodus = true

# Optionally output material properties to Exodus for visualization:

output\_material\_properties = 'porosity intragr\_release intergr\_release total\_fgr'

[]

In this example:

* We define temp as the temperature variable (which a heat conduction kernel would solve for).
* fission\_rate is defined (this might be set by a user object converting a power density to local fission rate, or by reading from a file).
* We include burnup as a variable for completeness (it could be updated via a Postprocessor or AuxKernel that integrates the fission\_rate over time to get burnup). However, in this simple usage we also pass burnup = 0.0 as a parameter to SciantixMaterial just to set initial burnup. If the simulation covers significant burnup, one could update the burnup variable over time and modify the material to use a coupled burnup instead of a fixed parameter.
* The SciantixMaterial is then configured under [Materials]:
  + type = SciantixMaterial refers to our C++ class.
  + temperature and fission\_rate are linked to the MOOSE variables temp and fission\_rate respectively.
  + burnup = 0.0 sets initial burnup (this would be non-zero if simulating a pre-irradiated fuel rod starting at some burnup).
  + sciantix\_input\_file is optionally provided. This file (for example, "sciantix\_input.txt") would contain the SCIANTIX model settings such as diffusion coefficients options, resolution rate correlations, initial grain size, initial porosity, etc., in whatever format SCIANTIX expects. If this parameter is left empty, the material assumes default model parameters in SCIANTIX (which might be hard-coded defaults).

After this setup, when you run the simulation, the SciantixMaterial will produce properties each time step:

* porosity, intragr\_release, intergr\_release, total\_fgr, swelling\_strain are computed. You can request them to be output (as shown, using output\_material\_properties for Exodus output), or use them in other calculations. For instance, if you have a solid mechanics model for the fuel, you could configure it to take swelling\_strain as an eigenstrain (in MOOSE’s TensorMechanics module, one can add an Eigenstrain material that reads the tensor from another material). In the input, that might look like:

[Modules/TensorMechanics/Master]

[./fuel\_element]

block = 'fuel'

eigenstrain\_names = 'swelling\_strain' # This tells the mechanics module to use our swelling tensor

[../]

[]

* This way, the gas-induced swelling is added to the mechanical strain tensor, causing expansion of the fuel.

**Compilation and Linking with the SCIANTIX Library**

To compile your MOOSE-based application with the external SCIANTIX library, a few steps are required:

1. **Build the SCIANTIX Library:** First, obtain the SCIANTIX source (from the official GitHub repo at sciantix-official). Follow its instructions to compile it as a library. Typically, this might involve using CMake or Make; the outcome should be either a static library (e.g., libsciantix.a) or a shared library (libsciantix.so) and associated headers. Ensure you note the install location of the library and headers.
2. **Include SCIANTIX in Your MOOSE App Build:**
   * **Headers:** Your MOOSE application needs to know where to find SCIANTIX header files. You can add the include path. In the MOOSE build system (which uses a hierarchical Make system by default), you can create a file like <YourApp>.mk in your application directory if it doesn’t exist. In that makefile, add a line to append the include directory. For example:

USER\_CXXFLAGS += -I/path/to/sciantix/include

* + This ensures that the compiler can locate Sciantix/Simulation.h or any other included headers. (Alternatively, MOOSE might provide a variable like USER\_INC\_DIR or similar; the key is to pass the -I flag for the Sciantix include path.)
  + **Library Linking:** Add the Sciantix library to the linking flags. Using the MOOSE Make system, you can append to ADDITIONAL\_LIBS. For example :

ADDITIONAL\_LIBS := /path/to/sciantix/lib/libsciantix.a

* + If the library is shared (.so or .dll on Windows), you might instead specify -L/path/to/sciantix/lib -lsciantix. Ensure any dependent libraries of Sciantix (if it requires other libs) are also linked.
  + The MOOSE build will then link your application with SCIANTIX. As Cody Permann noted, you do **not** need to modify MOOSE’s core makefiles; just your app’s makefile or a new .mk include will do . This keeps the customizations local to your app.
  + If your MOOSE version uses CMake (this might be the case for newer versions), the concept is similar: use target\_include\_directories and target\_link\_libraries in your app’s CMakeLists.txt to add Sciantix. For example:

include\_directories(/path/to/sciantix/include)

target\_link\_libraries(${APPLICATION\_NAME} PUBLIC /path/to/sciantix/lib/libsciantix.a)

* + (Replace with the actual target name and path.)

1. **Verify Linking and Runtime:** Once the above is set, compile your application (make -j or the equivalent CMake build). The compiler should find the Sciantix headers and compile SciantixMaterial.C successfully. Then the linker should produce an executable with Sciantix linked in. If Sciantix is a shared library and you linked dynamically, ensure the library is in your LD\_LIBRARY\_PATH or equivalent at runtime. Using a static library (.a) avoids runtime loading issues and is often simpler for coupling.
2. **Usage and Testing:** In your input file, use the new SciantixMaterial as shown. Run the simulation. It may be useful to start with a simple test (e.g., constant temperature and constant fission rate) to verify that intragr\_release, intergr\_release, etc., update over time in a plausible way (e.g., gas release starts at 0 and increases, swelling increases, etc.). Compare against standalone Sciantix calculations if possible for validation.

**Summary:** By following these steps, you effectively let MOOSE drive the SCIANTIX calculations: at each time step, MOOSE passes the current state of the fuel to SCIANTIX and receives updated fission gas release and swelling information . This tight integration harnesses SCIANTIX’s detailed physics models within a multiphysics simulation. Because SCIANTIX is open-source under MIT license , it is straightforward to incorporate it into MOOSE-based applications, enabling advanced fuel behavior modeling with relatively little additional code.