A new material model for magnetostrictive materials in the open-source Elmer FEM software

Juha Jeronen[1], Paavo Rasilo[1], Juhani Kataja[2]

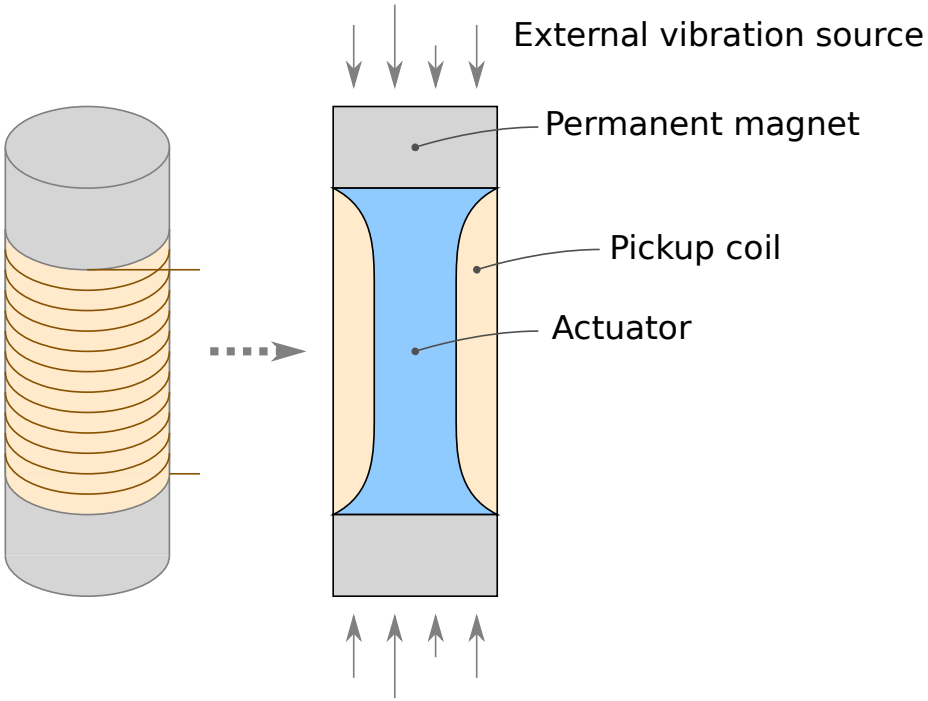
[1]Tampere University of Technology
[2]CSC IT Center for Science Ltd.

XIII Finnish Mechanics Days, 29.–31.8.2018
juha.jeronen@tut.fi

TAMPERE UNIVERSITY OF TECHNOLOGY

Magnetostriction

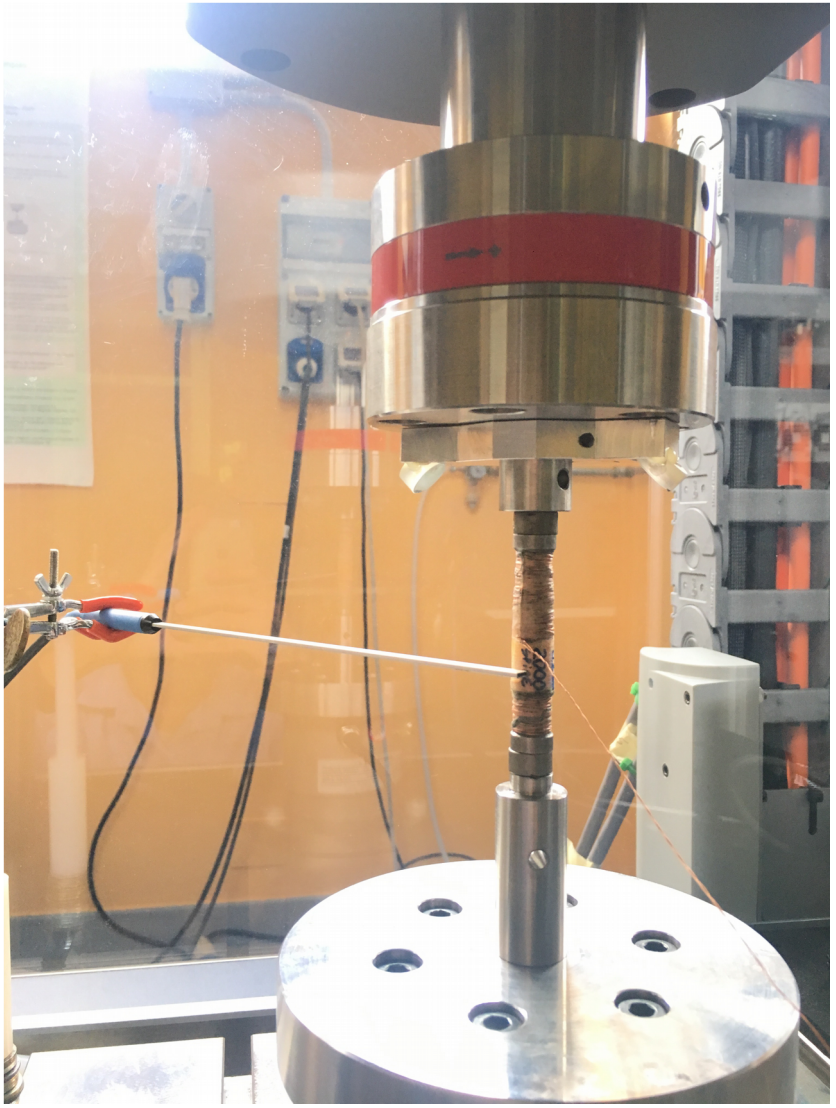
- Is a **magnetomechanical phenomenon**
  - *Joule effect* (1842):
    A property of ferromagnetic materials that causes them to change their shape when subjected to a magnetic field. [Wikipedia]
  - *Villari effect* (1865, *a.k.a. inverse magnetostrictive effect*, after Emilio Villari):
    Change of magnetic susceptibility of a material when subjected to a mechanical stress. [Wikipedia]
- Useful for **energy harvesting**:
  - With the help of a permanent magnet and a coil, utilize the Villari effect to extract a current when a magnetostrictive actuator is subjected to an external source of mechanical vibrations.



Giant magnetostrictive materials

- Typically these effects are very small; e.g.:
  - Iron, $r := \varepsilon_{magn} / \varepsilon_{mech}$ ~ O(1e-6)
- But **giant magnetostrictive materials** exist; e.g.:
  - **Terfenol-D** (1970s): the alloy $Tb_x Dy_{1-x} Fe_2$, where x ~ 0.3; r ~ O(1e-3)
  - **galfenol** (1998): generic name for gallium-iron alloys; r ~ O(1e-5)

    (*-nol*: Naval Ordnance Laboratory)



A galfenol rod in experimental setup for material characterization.
Image credit: Umair Ahmed, TUT

Governing equations

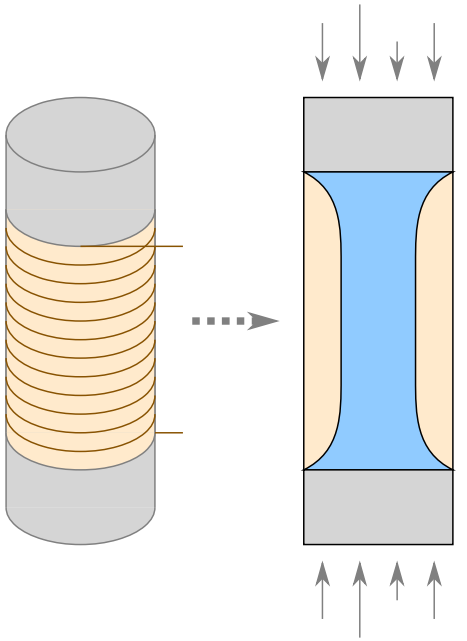
- Essentially, in the actuator:

$$\nabla \cdot \boldsymbol{\sigma} = \nabla \cdot \boldsymbol{\sigma}_0$$ *linear momentum (steady state)*

$$\nabla \times \mathbf{H} + \kappa \frac{\partial \mathbf{A}}{\partial t} = 0$$ *Ampere's and Faraday's laws*

- Elsewhere:

$$\nabla \times \mathbf{H} + \kappa \frac{\partial \mathbf{A}}{\partial t} = \mathbf{J}_\mathrm{s} + \nabla \times \mathbf{H}_\mathrm{c}$$



- $\boldsymbol{\sigma} = \boldsymbol{\sigma}(\boldsymbol{B}, \boldsymbol{\varepsilon})$: stress, $\boldsymbol{\sigma}_0$: prestress source term, $\boldsymbol{H} = \boldsymbol{H}(\boldsymbol{B}, \boldsymbol{\varepsilon})$: magnetic field strength, $\boldsymbol{A}$: magnetic vector potential, $\boldsymbol{H} = \nu \nabla \times \boldsymbol{A}$
- κ: electrical conductivity; $\kappa \neq 0$ in actuator and permanent magnets
- ν: reluctivity, in this study considered constant
- $\boldsymbol{J}_\mathrm{s}$: source current; $\boldsymbol{J}_\mathrm{s} \neq 0$ only in the pickup coil
- $\boldsymbol{H}_\mathrm{c}$: coercive field; $\boldsymbol{H}_\mathrm{c} \neq 0$ only in the permanent magnets

Material model

- In the actuator, a **constitutive law** couples $\boldsymbol{\sigma}$ and $\boldsymbol{H}$ in terms of a scalar potential, ψ:

$$\mathbf{H} = \frac{\partial \psi}{\partial \mathbf{B}} \qquad \boldsymbol{\sigma} = \frac{\partial \psi}{\partial \boldsymbol{\varepsilon}}$$

$$\psi = \tfrac{1}{2}\lambda [I_1]^2 + \mu I_2 + \sum_{k=1}^{n_\alpha} \alpha_k [I_4]^k + \sum_{k=1}^{n_\beta} \beta_k [I_5]^k + \sum_{k=1}^{n_\gamma} \gamma_k [I_6]^k$$

$$I_1 = \operatorname{tr} \boldsymbol{\varepsilon} \qquad I_2 = \operatorname{tr} \boldsymbol{\varepsilon}^2$$

$$I_4 = \mathbf{B} \cdot \mathbf{B} \qquad I_5 = \mathbf{B} \cdot \mathbf{e} \cdot \mathbf{B} \qquad I_6 = \mathbf{B} \cdot \mathbf{e} \cdot \mathbf{e} \cdot \mathbf{B}$$

rank-1

tensor contraction (on indices facing the operator)

- $\boldsymbol{B}$: magnetic flux density, $\boldsymbol{\varepsilon}$: Cauchy strain, ψ: Helmholtz free energy density
- $\boldsymbol{e}$: deviatoric strain, $\boldsymbol{e} = \boldsymbol{\varepsilon} - (1/3)\ \mathrm{tr}\ \boldsymbol{\varepsilon}$

- $I_1, \ldots, I_6$: scalar invariants (I_3 not needed because linear elasticity)

e.g. in any cartesian system, $I_6 = B_j\, e_{jk}\, e_{km}\, B_m$

- λ, μ: Lamé parameters (isotropic material)
- α_k, β_k, γ_k: empirical model fitting coefficients

Language matters

- Beside the material model, a **solver** is needed **for the governing equations**
- Elliptic 2nd-order PDEs, so FEM is an attractive option

- **Issue**: manually implementing FEM for a specific problem makes no sense
  (except once, as a learning exercise)
  - Level of abstraction of commonly used tools (e.g. linear algebra libraries) too far removed from the problem domain
  - This *impedance mismatch* in the language makes the specific solver much longer than it needs to be, and gives plenty of room for bugs to hide (concepts at an inappropriate level!)
  - Significant amount of work, very low re-usability; must repeat the process each time

- **Solution**: *bring the language closer to the problem domain*
  - Monumental task… fortunately, already done

- Strategy a): *general-purpose weak form compiler*
  - Allow the user to specify the problem in (or very close to) its mathematically native language
  - Choose finite element spaces, write weak forms, let the machine do the rest
  - FEniCS, Firedrake, FreeFEM++, COMSOL Multiphysics
  - Either extending an existing programming language (FEniCS, building on Python) or defining a new domain-specific language (DSL)

- Strategy b): *modular solver framework*
  - Library of functions (vs. language extension in a)) that packages parts of the solution process
  - Building blocks can be individual discretized terms (SfePy) or whole equations (Elmer)

Solution strategy

- **Elmer**: fast, scalable, modular collection of FEM solvers, written in Fortran, developed by CSC.
  - WhitneyAVSolver and linear elasticity (details in Elmer Models Manual, ElmerSolver Manual)
  - For geometry definition and meshing, Gmsh. For postprocessing, ParaView.

- Instead of implementing the material model by hand, we have built a **code generator**.
  - Program that writes programs. Eliminate human intervention in repetitive work.

  - Reduce the impedance mismatch; require less code from the human.

  - Decouple the material model definition from the implementation of the FEM code.

  - *Agile science: easily investigate a whole class of material models* – since much of the code can be auto-generated again whenever something is changed in the model definition.

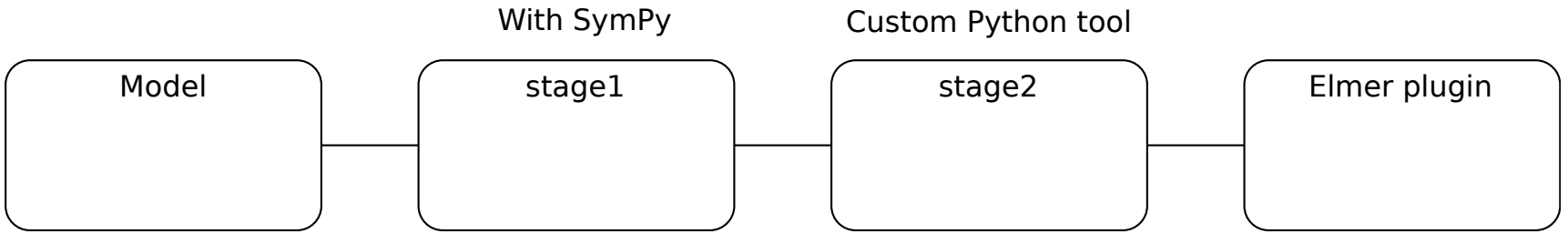
  - **Open source** (BSD 2-clause), published on GitHub: TUTElectromechanics/mm-codegen.

- Built on **Python 3**
  - Excellent library coverage; easy to learn; easy to find help online; rising language of science
  - Shares much of the Lisp advantage: very high abstraction level; short, expressive programs
  - *mm-codegen*: implemented in < 2e3 lines of Python (*including* comments and docstrings), outputs > 7e3 lines of Fortran (just *.f90*; 2× that if including *.h*) *per material model*

- Much of the material model is symbolic definitions, but for flexibility, can also import custom Fortran functions (and some subroutines).

Design

- *What do we need to consider when making something like this?*



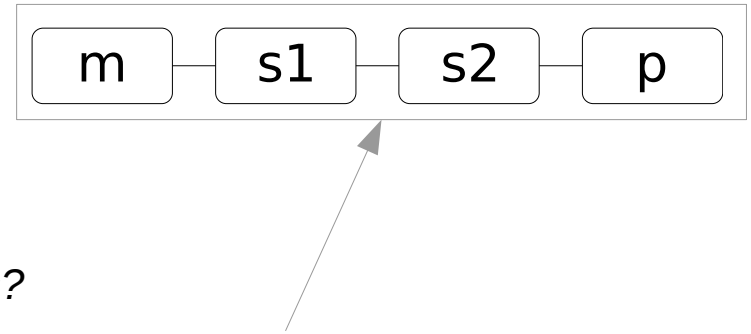
- Scalars easy to treat programmatically; process the math in component form.

- Two-stage process:

  - **s1**: Compile symbolic definitions into Fortran (*.f90*) pure functions.
    - Generate any implicit definitions (more on this later).
    - *Pure function*: return value depends only on arguments; also, no side effects.

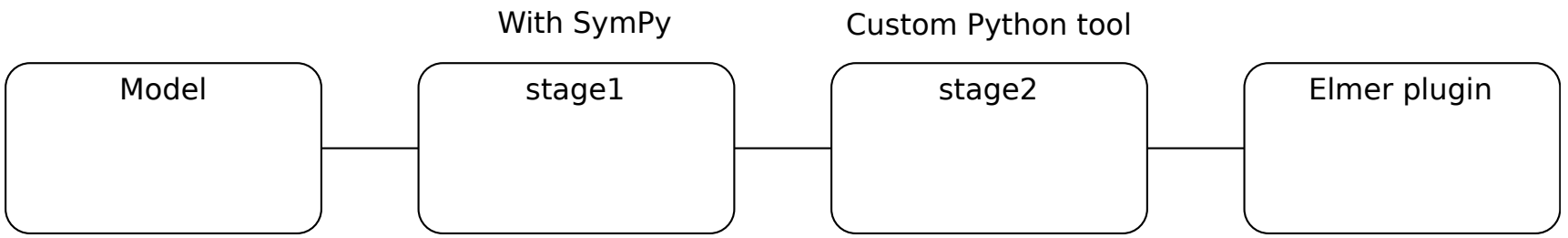
  - **s2**: Read output of *s1* and user-given custom codes; write public interface.
    - Fortran source-to-source transformation pass.
    - Public API: arguments include only independent variables and model parameters.

- Add a manually implemented wrapper to package the result into an Elmer plugin.
  - The wrapper works with any material model (that Elmer currently accepts) without needing any changes. It is specific only to Elmer's magnetostrictive solver.
  - If needed, it is easy to write similar wrappers for other solvers in Elmer.



Design

- *What do we need to consider when making something like this?*



- Scalars easy to treat programmatically; process the math in component form.

- Two-stage process:

  - **s1**: Compile symbolic definitions into Fortran (*.f90*) pure functions.
    - Generate any implicit definitions (more on this later).
    - *Pure function*: return value depends only on arguments; also, no side effects.

  - **s2**: Read output of *s1* and user-given custom codes; write public interface.
    - Fortran source-to-source transformation pass.
    - Public API: arguments include only independent variables and model parameters.

- Add a manually implemented wrapper to package the result into an Elmer plugin.
  - The wrapper works with any material model (that Elmer currently accepts) without needing any changes. It is specific only to Elmer's magnetostrictive solver.
  - If needed, it is easy to write similar wrappers for other solvers in Elmer.

Model

```
def let(k, v, s=True):
    defs[keyify(k)] = simplify(v) if s else v
Bs = sy.symbols("Bx, By, Bz")
εs = sy.symbols("εxx, εyy, εzz, εyz, εzx, εxy")
es = tuple(symutil.make_function(name, *εs)
        for name in ("exx", "eyy", "ezz", "eyz", "ezx", "exy"))
Is = tuple(symutil.make_function(name, *args)
                for name, args in (("I1", εs),
                                   ("I2", εs),
                                   ("I4", Bs),
                                   ("I5", Bs + es),
                                   ("I6", Bs + es)))
ψ = symutil.make_function("ψ", *Is)  # Helmholtz f.e.d.

B = sy.Matrix(Bs)                  # magnetic flux density
ε = symutil.voigt_to_mat(εs)       # Cauchy strain
e = symutil.voigt_to_mat(es)       # deviatoric strain
I1, I2, I4, I5, I6 = Is

ε_vol = sy.factor(sy.S("1/3") * ε.trace())
e_def = ε - ε_vol * sy.eye(3)
for _, (r, c) in symutil.voigt_mat_idx():
    let(e[r, c], e_def[r, c], s=False)
for key, val in ((I1, ε.trace()),
                 (I2, (ε.T * ε).trace())):
    let(key, val)
for key, val in ((I4, B.T * B),
                 (I5, B.T * e * B),
                 (I6, B.T * e * e * B)):
    let(key, val[0,0])  # unwrap the scalar
```

```
λ, μ = sy.symbols("λ, μ")
ψ_mech = sy.S("1/2") * λ * I1**2  +  μ * I2  # lin. elasticity

i0, nα, nβ, nγ = 1, 11, 1, 1
*αs, = sy.symbols("α{:d}:{:d}".format(i0, i0+nα))  # α1, .., α11
*βs, = sy.symbols("β{:d}:{:d}".format(i0, i0+nβ))
*γs, = sy.symbols("γ{:d}:{:d}".format(i0, i0+nγ))
I4_terms = sum(αi * I4**i for i, αi in enumerate(αs, start=i0))
I5_terms = sum(βi * I5**i for i, βi in enumerate(βs, start=i0))
I6_terms = sum(γi * I6**i for i, γi in enumerate(γs, start=i0))
ψ_magn = I4_terms + I5_terms + I6_terms

let(ψ, ψ_mech + ψ_magn)
```

- Separate *the functional dependency chains* from *the actual definitions* (important later, in **s1**).
- Can use SymPy for symbolic computation.
  - *Definitions, instructions; actual math automatic.*
- Still looks like a program, but *somewhat* close to the math (in the original Fortran spirit).
- Python 3 allows Unicode variable names.
  - For easily entering them, latex-input.
- *potentialmodelbase.py*, *polymodel.py* [minor edits]

From model to Fortran

- SymPy has sympy.utilities.codegen, which takes symbolic expressions and outputs Fortran 90.
  - Missing some features we need; but still less work for us if we take advantage of it.
  - Sufficient for a simple first stage, which puts all symbols from the RHS of a definition into the formal parameter list of the function being generated.

- *Recursive pool of definitions*: a definition in the model can refer to any others (except no loops).
  - Since the RHSs are simple expressions with no conditionals or other forms of control flow, preventing mutual recursion *guarantees* absence of infinite loops in the generated code.
  - Easy to enforce by keeping track of dependencies already seen when (recursively) processing any given definition.

- *Derivatives automatically defined if needed.*
  - In the Helmholtz free energy density formulation above, the independent variables were $\boldsymbol{B}$, $\boldsymbol{\varepsilon}$, and on top of them, we constructed $\boldsymbol{e} = \boldsymbol{e}(\boldsymbol{\varepsilon})$, $I_k = I_k(\boldsymbol{B}, \boldsymbol{\varepsilon})$, and $\psi = \psi(I_1, I_2, I_4, I_5, I_6)$.
  - Hence we can obtain $\boldsymbol{H} = \partial\psi / \partial\boldsymbol{B}$ (and similarly $\boldsymbol{\sigma} = \partial\psi / \partial\boldsymbol{\varepsilon}$) via the chain rule:

$$\frac{\partial \psi}{\partial B_j} = \frac{\partial \psi}{\partial I_k} \frac{\partial I_k}{\partial B_j}$$

  - If needed, we recurse on the RHS (*given the dependency chain, this is automatic in SymPy*).
    - In this particular case, we stop here, since the I_k directly depend on the B_j.
    - In the case of e.g. $\partial I_4 / \partial \varepsilon_{xx}$, we do one more step, because $I_4 = I_4(\boldsymbol{B}, \boldsymbol{e})$, and $\boldsymbol{e} = \boldsymbol{e}(\boldsymbol{\varepsilon})$.
    - For more complex models, the final chain rule expressions become very long.

From model to Fortran

- In the model, we just add this to the pool (hence telling **s1** we want a Fortran function for it):

$$\frac{\partial \psi}{\partial B_j} = \frac{\partial \psi}{\partial I_k} \frac{\partial I_k}{\partial B_j}$$

where the RHS uses *unknown functions* (known dependency chain; no explicit expression).

- When scanning the pool of definitions, **s1** notices that here the RHS contains some derivatives. This triggers a symbolic derivative generator:

$$\frac{\partial \psi}{\partial B_j} = \frac{\partial \psi}{\partial I_k} \frac{\partial I_k}{\partial B_j}$$

Reduce operation count by re-grouping terms in a semi-intelligent fashion.

If I_k defined, derive this from its definition, optimize, add result to pool.

If ψ defined, derive this from its definition, optimize, add result to pool.

- **s1** then recurses into the new definitions, until no more derivatives appear on the RHS.
- Any derivatives already in the pool of definitions are re-used, to avoid unnecessary work.

- If the definitions show some particular derivative to be identically zero, that term is deleted (via symbolic substitution, ensuring correct math) from all RHS expressions that reference it.
- Hence zeros are recognized based on not only the structure (e.g. $\partial I_1 / \partial B_x \equiv 0$, since $I_1 = I_1(\boldsymbol{\varepsilon})$), but also on each particular RHS (e.g. $\partial^2 I_1 / \partial \varepsilon_{xx}^2 \equiv 0$, since the definition is $I_1 = \mathrm{tr}\ \boldsymbol{\varepsilon}$).

From model to Fortran

- Even when dealing with pure functions, ***resist the temptation to substitute***.
  - Unless each symbol appears only once on the RHS, substituting its definition will introduce *common subexpressions (CSE)*, leading to unnecessary re-computation. (Unless the optimizer is smart enough to notice and generate temporaries for CSEs; but why risk it?)
  - *Example*. Consider the definitions

    $y = a + b\,x, \quad z = y^2$

    where *a*, *b*, *x* are free. If we want only *z*, substitution gives us (in executable pseudocode)

    ```
    def z(a, b, x):
        return (a + b*x) * (a + b*x)
    ```

    which unnecessarily repeats the calculation (a + b*x). Furthermore, once the expressions become more complex, the generated code becomes completely unreadable for humans.
  - To obtain clean, well-optimized code, **s1** keeps the structure of the definitions intact:

    ```
    def y(a, b, x):
        return a + b*x
    def z(y):
        return y * y
    ```

    Issue: Now the caller must supply a value for *y*.

Source-to-source transformation

- **Solution**: a second stage. Given the first stage output:

```
def y(a, b, x):
    return a + b*x
def z(y):
    return y * y
```

we analyze it and generate more code to provide the interface we want:

```
def z_public(a, b, x):
    y_ = y(a, b, x)
    return z(y_)
```

Now free arguments only.

y and *z* are *"s1 functions"*, i.e., code generated by stage 1.

- This is the job of **stage 2**. It works essentially by generating dependency trees.
  - Its input is pure functions, so with a naming convention, the argument names specify which other s1 functions each s1 function needs to be able to compute its output.
  - Analyze recursively; collect arguments into **bound** and **free** sets.
  - Copy any **free** arguments to the argument list of the public API function being written.
  - Automatically compute any **bound** arguments by writing code to call other *s1* functions (in the appropriate order to resolve any dependencies, recursively).
    - Generate a temporary for each unique result to avoid any unnecessary re-evaluation.
  - Custom user-given pure functions can also be imported into the set of *s1* functions.
  - Subroutines may appear at the top level (i.e. not as a dependency), to allow array packing.

The Elmer plugin

```
! s1 generated code (blank lines omitted for brevity):
REAL(KIND=dp) function I6(Bx, By, Bz, exx, exy, eyy, eyz, ezx, ezz)
use types
implicit none
I6 = Bx**2*(exx**2 + exy**2 + ezx**2) + 2*Bx*(By*(exx*exy + exy*eyy + &
     eyz*ezx) + Bz*(exx*ezx + exy*eyz + ezx*ezz)) + By**2*(exy**2 + &
     eyy**2 + eyz**2) + 2*By*Bz*(exy*ezx + eyy*eyz + eyz*ezz) + Bz**2* &
     (eyz**2 + ezx**2 + ezz**2)
end function
```

Generated from the model.

```
! corresponding s2 generated code (blank lines omitted for brevity):
REAL(KIND=dp) function I6_public(Bx, By, Bz, epsxx, epsxy, epsyy, epsyz, &
     epszx, epszz)
use types
implicit none
REAL(KIND=dp) exx_
! ...snip...
exx_ = exx(epsxx, epsyy, epszz)
exy_ = exy(epsxy)
eyy_ = eyy(epsxx, epsyy, epszz)
eyz_ = eyz(epsyz)
ezx_ = ezx(epszx)
ezz_ = ezz(epsxx, epsyy, epszz)
I6_public = I6(Bx, By, Bz, exx_, exy_, eyy_, eyz_, ezx_, ezz_)
end function
```

Generated from the **s1** code.

The Elmer plugin

```
! user definitions (additional s1 functions/subroutines)
! σ = ∂ϕ/∂ε
subroutine S(dphi_depsxx, dphi_depsyy, dphi_depszz, &
             dphi_depsyz, dphi_depszx, dphi_depsxy, &
             S_out)
use types
implicit none
REAL(KIND=dp), intent(in) :: dphi_depsxx
REAL(KIND=dp), intent(in) :: dphi_depsyy
REAL(KIND=dp), intent(in) :: dphi_depszz
REAL(KIND=dp), intent(in) :: dphi_depsyz
REAL(KIND=dp), intent(in) :: dphi_depszx
REAL(KIND=dp), intent(in) :: dphi_depsxy
REAL(KIND=dp), intent(out), dimension(1:3, 1:3) :: S_out

S_out(1, 1) = dphi_depsxx
S_out(1, 2) = dphi_depsxy
S_out(1, 3) = dphi_depszx
S_out(2, 1) = dphi_depsxy ! symm.
S_out(2, 2) = dphi_depsyy
S_out(2, 3) = dphi_depsyz
S_out(3, 1) = dphi_depszx ! symm.
S_out(3, 2) = dphi_depsyz ! symm.
S_out(3, 3) = dphi_depszz

end subroutine
```

Human-written Fortran, treated on equal footing with any other s1 code.

The **only** things this subroutine does:

- Declare the s1 functions whose return values it wants (by name, in the formal parameter list),
- Pack the data into an output array.

The Elmer plugin

```
! corresponding s2 generated code (140 lines; severely snipped to fit here):
subroutine S_public(Bx, By, Bz, epsxx, epsxy, epsyy, epsyz, epszx, epszz, &
    S_out)
use types
implicit none
REAL(KIND=dp), intent(in) :: Bx  ! ...
REAL(KIND=dp), intent(out), dimension(1:3, 1:3) :: S_out
REAL(KIND=dp) I1_  ! ...

I1_ = I1(epsxx, epsyy, epszz)  ! ...
exx_ = exx(epsxx, epsyy, epszz)  ! ...
dI6_dexx_ = dI6_dexx(Bx, By, Bz, exx_, exy_, ezx_)
dexx_depsxx_ = dexx_depsxx()  ! ...
dphi_dI1_ = dphi_dI1(I1_, lam_)
dphi_dI2_ = dphi_dI2(mu_)
dphi_dI5_ = dphi_dI5(bet1_)
dphi_dI6_ = dphi_dI6(gam1_)
dphi_depsxx_ = dphi_depsxx(dI1_depsxx_, dI2_depsxx_, dI5_dexx_, dI5_deyy_, &
    dI5_dezz_, dI6_dexx_, dI6_deyy_, dI6_dezz_, dexx_depsxx_, &
    deyy_depsxx_, dezz_depsxx_, dphi_dI1_, dphi_dI2_, dphi_dI5_, &
    dphi_dI6_)
dphi_depsxy_ = ! ...

call S(dphi_depsxx_, dphi_depsyy_, dphi_depszz_, dphi_depsyz_, dphi_depszx_, &
    dphi_depsxy_, S_out)

end subroutine
```

Output args are placed last automatically.

All the magic to satisfy the requirements happens in this generated code.

Temporaries are computed only once also when multiple components of $\boldsymbol{\sigma}$ (i.e. arguments of S) need the same quantity.

Note argument order different from what *S_public* itself takes.
The correct order is read from the interfaces input to **s2**.

The Elmer plugin

```
! The Elmer plugin (human-written, API adaptor for Elmer's magnetostriction solver):
subroutine mgs_S(B1, B2, B3, e11, e12, e13, e21, e22, e23, e31, e32, e33, sigma_out)
  use types
  use MgsContainer
  implicit none
  REAL(KIND=dp), intent(in) :: B1, B2, B3, e11, e12, e13, e21, e22, e23, e31, e32, e33
  REAL(KIND=dp), intent(out), dimension(1:3, 1:3) :: sigma_out
  call S_public(B1, B2, B3, e11, e12, e22, e23, e13, e33, sigma_out)
end subroutine
```

```
! container (human-written)
module MgsContainer
contains
#include "./generated/mgs_poly_impl.f90"
#include "./mgs_physfields.f90"
#include "./generated/mgs_poly.f90"
end
```

This just loads all the definitions into a namespace for the plugin.

```
#!/bin/bash
# Compile script for the Elmer plugin
elmerf90 mgs-container.F90 mgs_plugin.f90 -o galfenol.so
```

Discussion

- Full Jacobian and Hessian of ψ w.r.t. the independent variables generated automatically.
- Can define e.g. ψ as a custom Fortran code (must then provide also its immediate derivatives).

- In **s1**, the generated code is fairly performance-optimized as-is (minimizing operation count):
  - Identically zero derivatives are deleted,
  - Expressions are optimized symbolically,
  - Unnecessary re-computation is avoided.

- In **s2**, each top-level (public API) call generates one function call per unique bound argument discovered in the dependency tree.
  - Dependency resolution and temporary generation makes this *exactly* one (no more).
  - Function calls in Fortran are cheap (compared to Python).
    - Expensive function calls were a myth already in 1977 [Steele].
    - The Fortran compiler may inline any parts of the code it wants.
      - E.g. to eliminate calls to functions that just return a constant.
      - Much easier for the compiler to inline code than to find CSEs to extract into functions.

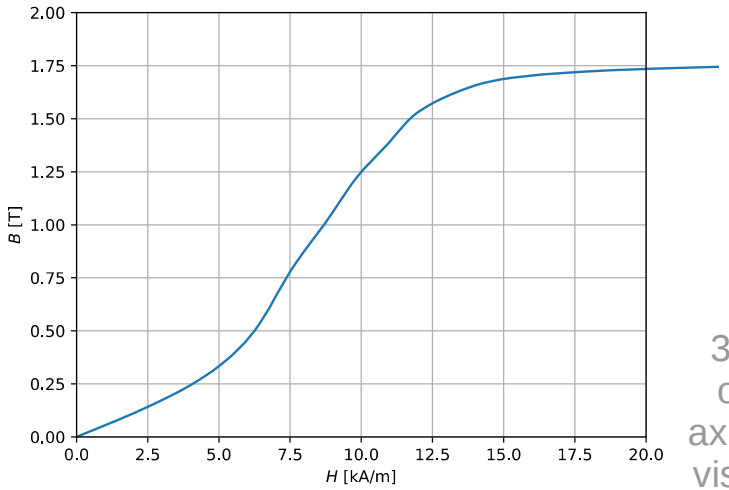
- The structured approach retains information about the structure of each expression.
  - May be useful as optimization advice for the Fortran compiler.

- *Math is cheap, bandwidth expensive*; register and/or cache pressure from the **s2** temporaries?

- The generated code is (arguably) highly readable.
  - Layout could be improved, to group terms like a mathematician would.

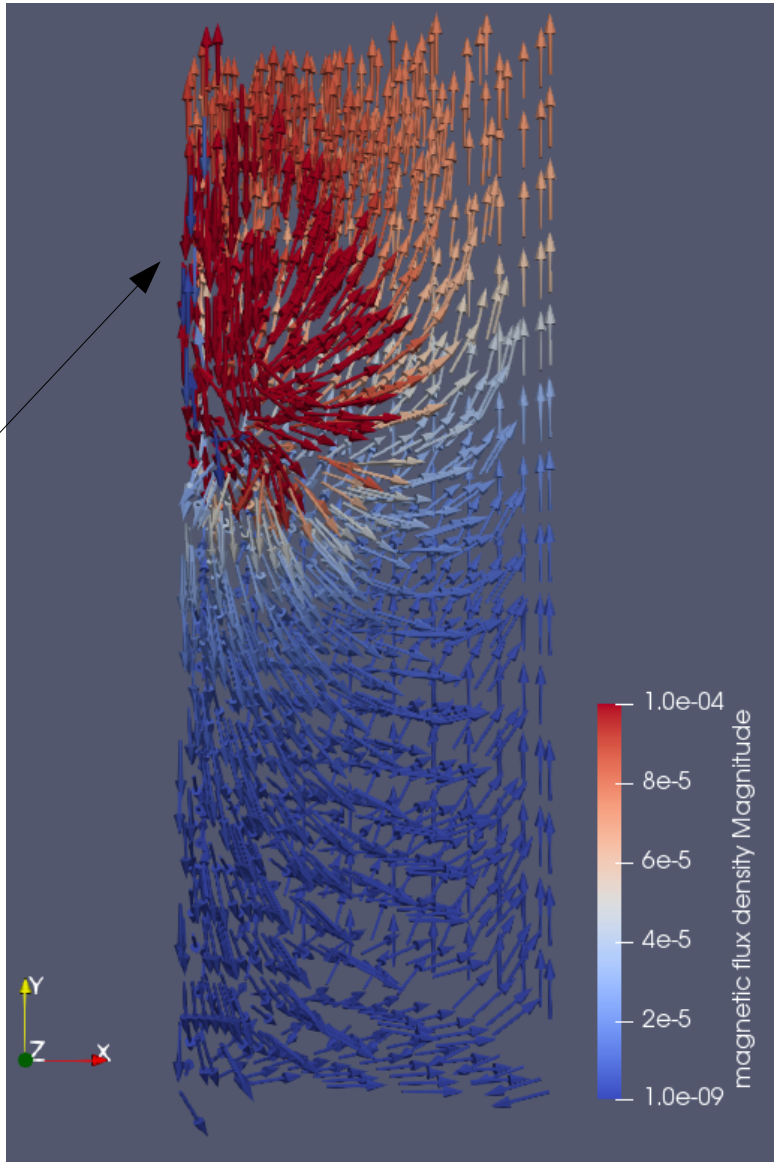
Results & future directions

- Works, looks promising, some tuning still needed
  - Generated code should be bug-free; highly modular, very simple compiler, programmed in functional style [Hughes 1984], with aggressive input validation
  - BH curve computed by model looks reasonable
- Validation needed for Elmer's magnetostriction solver
  - Can compare results to axisymmetric 2D MATLAB and COMSOL models for our test case
- Possible to try other variants for ψ
  - E.g. B-spline based piecewise polynomial



(to saturation)

Galfenol actuator



3D steady-state simulation of a one-eight model of an axisymmetric case in Elmer, visualization with ParaView.

Conclusion

- A new material model for magnetostrictive materials was implemented in Elmer.
- Instead of implementing the model directly, a fairly general Fortran code generator was implemented in Python.
  - Easy to investigate a whole class of material models by changing the model definition and recompiling.
  - Modular; can be customized to perform other similar tasks that require generating Fortran code from symbolic definitions.
  - Open source; available on GitHub to anyone interested.
- Techniques well known in computer science and in information technology have much to offer to computational science.
  - Especially software automation via symbolic computation, code generation.
  - Functional programming helps reduce bugs and increase modularity and maintainability.

Thank you!

- https://github.com/TUTElectromechanics/mm-codegen
  - Model definition and code generator.
  - *Includes a copy of these slides, with clickable links.*

- https://github.com/juhanikataja/magnetostriction-days
  - Magnetostriction solver for Elmer.

- Fonteyn, K., Belahcen, A., Kouhia, R., Rasilo, P., Arkkio, A. 2010. FEM for Directly Coupled Magneto-Mechanical Phenomena in Electrical Machines*, IEEE Trans. Magn.* **46**(8), 2923–2926.