

SLOTH TUTORIAL · COEFFICIENTS

Coefficients with Sympy



■ Why coefficients

The physics lives in Coefficient objects

1

Anatomy of a Coefficient

$$\frac{\partial \varphi}{\partial t} = -M \left[\omega W'(\varphi) - \lambda \Delta \varphi \right]$$

2

A first coefficient

M · mobility

Glossary::Mobility

ω W(φ) · free energy

Glossary::FreeEnergy
SLOTH uses its derivatives W', W''

λ · capillarity

Glossary::Capillary

3

Several fields

The integrator knows the equation

AllenCahn assembles the weak form, the residual and the Jacobian.

You provide the coefficients

formulas for the energy and the kinetics; Sympy writes the C++ and the exact derivatives.

4

In the problem

■ Anatomy

A Coefficient = quantity + scheme + function

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

```
Coefficient energy(Glossary::FreeEnergy, Scheme::Implicit, W(omega));
```

Quantity

Glossary::... says **what** it is:
FreeEnergy, Mobility, Capillary,
GradEnergy, Temperature, Dirichlet,
Neumann...

Integrators find a coefficient **by its quantity**.

Scheme

Implicit · current Newton iterate,
derivatives in the Jacobian

Explicit · fields at t^n , no derivative

SemiImplicit · both, see
explicit_variables

No scheme for a constant.

Function

a constant, a SLOTH built-in such as
 $W(\omega)$, or **your own class**
generated by Sympy from a JSON
file.

The topic of this tutorial.

■ Where the function comes from

Three ways to give the function

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

A constant

```
Coefficient c(  
  Glossary::Capillary,  
  lambda);
```

No scheme; derivatives are zero.

A SLOTH built-in

```
W(omega)  
Fw(omega)  
H()  
Log(a)  
GradientEnergy(lambda)
```

Double wells, interpolation, log energy, gradient energy. Generated by Sympy from SLOTH's own JSON file.

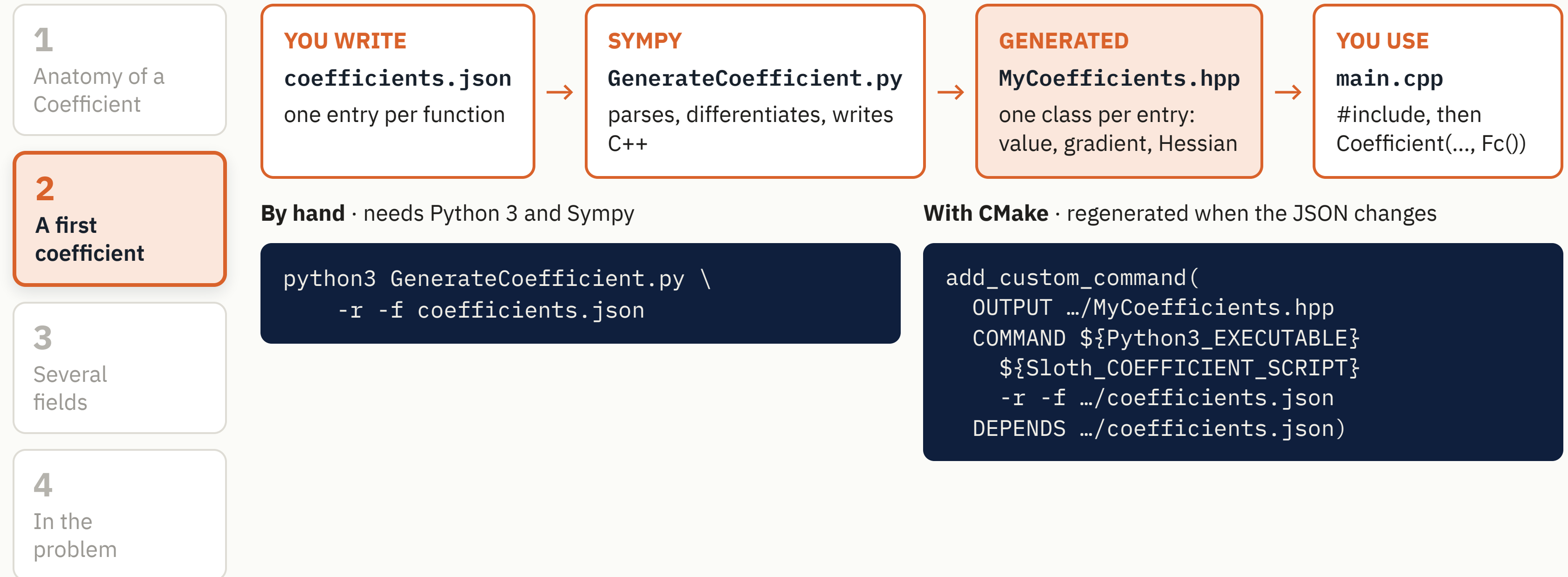
Your own formula

```
{  
  "expression": "...",  
  "variables": "c",  
  "class_name": "Gc",  
  "outputfile": "MyCoefficients"  
}
```

Sympy writes the class Gc and its derivatives; then `Coefficient(..., Gc())`.

■ The Sympy interface

From a formula to a C++ class



■ A first coefficient

The double well, written by you

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

```
[{
  "expression": "phi*phi*(1-phi)*(1-phi)",
  "variables": "phi",
  "class_name": "DoubleWell",
  "outputfile": "MyCoefficients"
}]
```

$$W(\varphi) = \varphi^2 (1 - \varphi)^2$$

expression	the formula, in Sympy syntax
variables	the unknowns; derivatives are taken w.r.t. them
class_name	name of the generated C++ class
outputfile	basename of the generated .hpp

constants, auxiliary_variables, coordinates:
optional, see next slides

```
#include "MyCoefficients.hpp"
Coefficient energy(Glossary::FreeEnergy,
  Scheme::Implicit, DoubleWell(omega));
```

omega is the **prefactor**: it multiplies the function and all its derivatives (default 1).

■ What Sympy writes

The generated class: value, gradient, Hessian

1

Anatomy of a
Coefficient

2

A first
coefficient

3

Several
fields

4

In the
problem

MyCoefficients.hpp · DoubleWell (extract)

```
class DoubleWell : public FunctionCoefficient { ... };

// value
double phi = input_vector[0];
double F = std::pow(phi, 2) * std::pow(1 - phi, 2);
return this->prefactor_ * F;

// first derivatives, one per variable
gradient[0] = this->prefactor_ *
    (std::pow(phi, 2) * (2 * phi - 2)
    + 2 * phi * std::pow(1 - phi, 2));

// second derivatives
hessian[0] = this->prefactor_ * (...);
```

Exact derivatives

for the Newton Jacobian, computed
once by Sympy, not by finite
differences.

Unknowns by position

input_vector[i] ← variables,
auxiliary_vector[i] ← auxiliary
variables, in declaration order.

Never edit it

change the JSON and regenerate; -r
overwrites the header.

■ Constants

Constants, physical constants and time

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

```
{  
  "expression": "a*x*x + b*x*y + c*z*x*y",  
  "variables": "x, y z",  
  "constants": "(a:2.0),(b:30.0),(c:0.1)",  
  "class_name": "FunctionA", ...  
}
```

Time-dependent coefficient

```
{  
  "expression": "a*x*x*t",  
  "variables": "x",  
  "constants": "(a:2.0),(t:T)",  
  "class_name": "FunctionC", ...  
}
```

A value or a SLOTH constant

H Planck constant · Physical::H

NA Avogadro constant · Physical::NA

K Boltzmann constant · Physical::K

R molar gas constant · Physical::R

T current time, updated by SLOTH

Constant or prefactor?

a constant is fixed when the class is generated;
the prefactor is chosen in main.cpp, for each Coefficient.

■ Several fields

Variables and auxiliary variables

1

Anatomy of a
Coefficient

2

A first
coefficient

3

Several
fields

4

In the
problem

$$f(c, \eta_1, \eta_2) = A c^2 (1 - c)^2 + B \left[c^2 + 6(1 - c) \sum_i \eta_i^2 - 4(2 - c) \sum_i \eta_i^3 + 3 \left(\sum_i \eta_i^2 \right)^2 \right]$$

Running example: two sintering particles. Cahn-Hilliard solves the density **c**, Allen-Cahn solves the grain fields **η₁, η₂**.
The same energy *f* enters both problems.

variables

the unknowns of the problem **that uses** the coefficient

Sympy differentiates with respect to them

read from the problem's VARS, in order

auxiliary_variables

fields solved by **another** problem

only read: no derivative

read from the problem's auxiliary variables, in order

■ Several fields

One energy, one class per problem

1

Anatomy of a Coefficient

for Cahn-Hilliard

```
{"expression": "A*c*c*(1-c)*(1-c) + B*(...)",  
  "variables": "c",  
  "auxiliary_variables": "eta(1..2)",  
  "class_name": "Fc", ...}
```

for Allen-Cahn

```
{"expression": "A*c*c*(1-c)*(1-c) + B*(...)",  
  "variables": "eta(1..2)",  
  "auxiliary_variables": "c",  
  "class_name": "Feta", ...}
```

2

A first coefficient

```
TransientPB ch_pb("CahnHilliard", ch_oper,  
  ch_vars,    // c, mu    → input_vector  
  {ch_coef, ch_coef}, ch_pst,  
  ac_vars);  // η1, η2 → auxiliary_vector
```

```
TransientPB ac_pb("AllenCahn", ac_oper,  
  ac_vars,    // η1, η2 → input_vector  
  {ac_coef, ac_coef}, ac_pst,  
  ch_vars);  // c, mu    → auxiliary_vector
```

3

Several fields



Ordering convention. Variables in the JSON follow the order of the problem's VARS; auxiliary variables follow the order of its auxiliary VARS. A wrong order is not detected: the wrong field is read.

4

In the problem

■ Several fields

N fields: indexed variables and sums

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

```
{
  "expression": "Sum( -(x[i]*x[i])/2.0
                    + (x[i]*x[i]*x[i]*x[i])/4.0,
                    (i, 1, 30) )",
  "variables": "x(1..30)",
  "class_name": "Gw",
  "outputfile": "GrainsCoefficients"
}
```

$$\sum_{i=1}^{30} \left(\frac{1}{4} x_i^4 - \frac{1}{2} x_i^2 \right)$$

Declare a family

$x(1..30)$ declares $x_1 \dots x_{30}$, in variables or auxiliary_variables.

Sum over it

$\text{Sum}(\text{expr}, (i, 1, 30))$ with $x[i]$ inside; several index ranges for double sums.

Consistent bounds

the bounds of the sum must match the declaration.

From 2 to N grains

in the sintering energy, $\text{Sum}(\text{eta}[i]*\text{eta}[i], (i, 1, 2))$: change 2 into N.

■ Several fields

Gradient energies: "gradient": true

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

```
{  
  "expression": "2.5*dot(x,x) + sdot(eta(1..2))",  
  "variables": "x",  
  "auxiliary_variables": "eta(1..2)",  
  "class_name": "GradC",  
  "outputfile": "SinteringCoefficients",  
  "gradient": true  
}
```

$$\frac{5}{2} \nabla c \cdot \nabla c + \sum_{i=1}^2 \nabla \eta_i \cdot \nabla \eta_i$$

```
Coefficient ch_grad(Glossary::GradEnergy,  
  Scheme::Implicit, GradC());
```

Each variable is a gradient

with gradient: true, x stands for ∇c and eta1 for $\nabla \eta_1$.

dot(x,x)

the scalar product $\nabla x \cdot \nabla x$

sdot(eta(1..2))

the sum $\sum \nabla \eta_i \cdot \nabla \eta_i$

Limitation

not combinable with coordinates.

■ Several fields

A field-dependent mobility

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

$$D(c, \eta) = D_{vol} \phi(c) + D_{vap} (1 - \phi(c)) + D_{surf} c(1 - c) + D_{gb} \sum_{i \neq j} \eta_i \eta_j$$

$$\phi(c) = c^3 (10 - 15c + 6c^2)$$

```
"expression": "dvol*c*c*c*(10.0-15.0*c+6.0*c*c) + ...",  
"variables": "c",  
"auxiliary_variables": "eta(1..2)",  
"constants": "(dvol:0.04),(dvap:0.002),  
              (dsurf:16.0),(dgb:1.6)",  
"class_name": "D"
```

```
Coefficient ch_mobility(Glossary::Mobility,  
                        Scheme::Explicit, D());
```

Volume, vapour, surface, grain boundary

four diffusion paths weighted by the fields.

Scheme::Explicit

evaluated with the fields of the previous time step; no derivative in the Jacobian.

■ In the problem

Coefficients, one collection per variable

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

```
// Cahn-Hilliard (sintering test)
```

```
Coefficient ch_energy(Glossary::FreeEnergy, Scheme::Implicit, Fc());
```

```
Coefficient ch_grad(Glossary::GradEnergy, Scheme::Implicit, GradC());
```

```
Coefficient ch_capillary(Glossary::Capillary, ch_lambda);
```

```
Coefficient ch_mobility(Glossary::Mobility, Scheme::Explicit, D());
```

```
Coefficients ch_coef(ch_energy, ch_capillary, ch_mobility, ch_grad);
```

```
TransientPB ch_pb("CahnHilliard", ch_oper, ch_vars, {ch_coef, ch_coef}, ch_pst, ac_vars);
```

Found by quantity

the integrator picks FreeEnergy, Mobility, Capillary... in the collection; their order inside is free.

One collection per variable

{ch_coef, ch_coef}: as many as VARS, in the same order (c, mu).

Which quantities?

each integrator lists the ones it reads on the PDEs page.

■ In the problem

Going further

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem

Space-dependent

```
"expression": "-pi*t*cos(pi*x)",  
"coordinates": "x,y",
```

boundary values or sources; attach to edges with `set_bdr_index_coef`.

Conditional

```
"expressions": [  
  {"expression": "...", "lower": 0.0, "upper": 0.75},  
  {"expression": "...", "default": true}],  
"range_variable": "x1",
```

piecewise formulas on one variable.

Semi-implicit

```
"variables": "x(1..30)",  
"explicit_variables": "xn(1..30)",
```

x_n at t^n ; use `Scheme::SemiImplicit`.

Combined

```
auto P = ProductCoefficient({&A, &B});  
auto S = SumCoefficient({&P, &P});
```

products and sums of generated classes.

■ In the problem

Six checks before running

1

Anatomy of a Coefficient

2

A first coefficient

3

Several fields

4

In the problem



Variable order

JSON variables = order of the problem's VARS



Auxiliary order

JSON auxiliary_variables = order of the auxiliary VARS



Every symbol declared

once, as a variable, auxiliary, constant or coordinate



Regenerate

after any change to the JSON (CMake does it)



The right scheme

Implicit for energies; Explicit to freeze at t^n ;
SemiImplicit with explicit_variables



The right quantity

the Glossary type the integrator expects

Where to find information

Going on from here

 **MANUAL**
Read

Coefficients

every JSON key and option

PDEs

quantities read by each
integrator

 **SETUP**
Install

Sympy

pip install sympy

Using an installed SLOTH

CMake custom command

 **TESTS**
Copy

Studies/sintering/test1

energy, gradient, mobility

HeatTransfer/2D/test5

coordinates

Coefficients/test2

constants

 **NEXT**
Go on

Input files in a nutshell

the whole main.cpp

Examples

commented cases