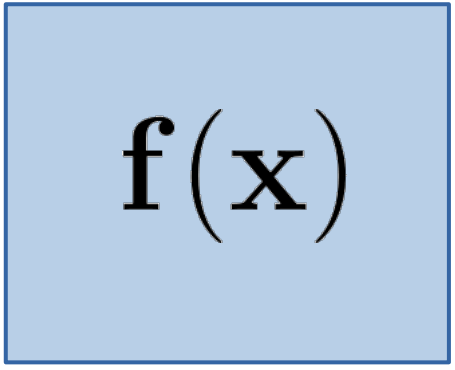
Automatic and symbolic differentiation primer

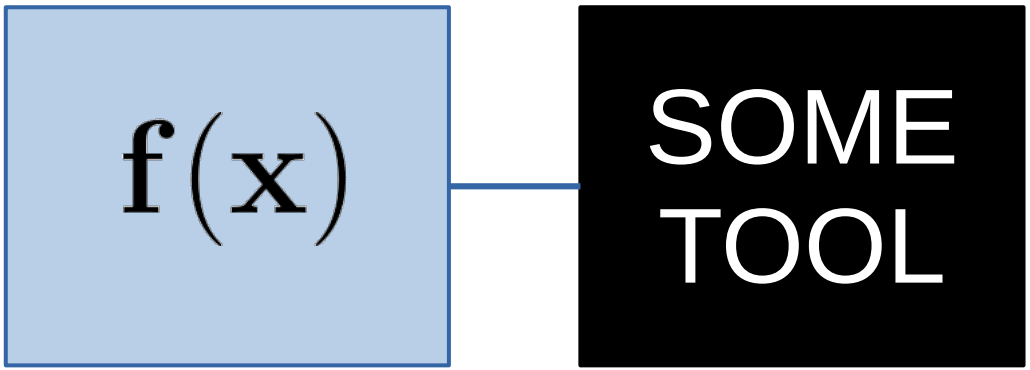
deal.II user and developer workshop
August 2024

Jean-Paul Pelteret

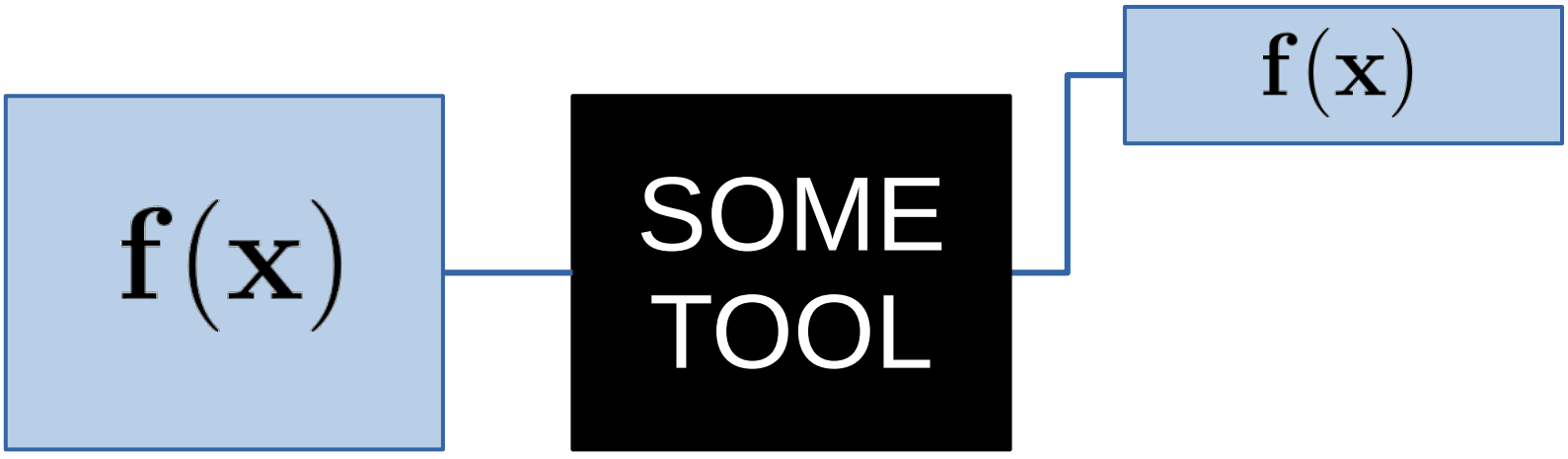
The idea



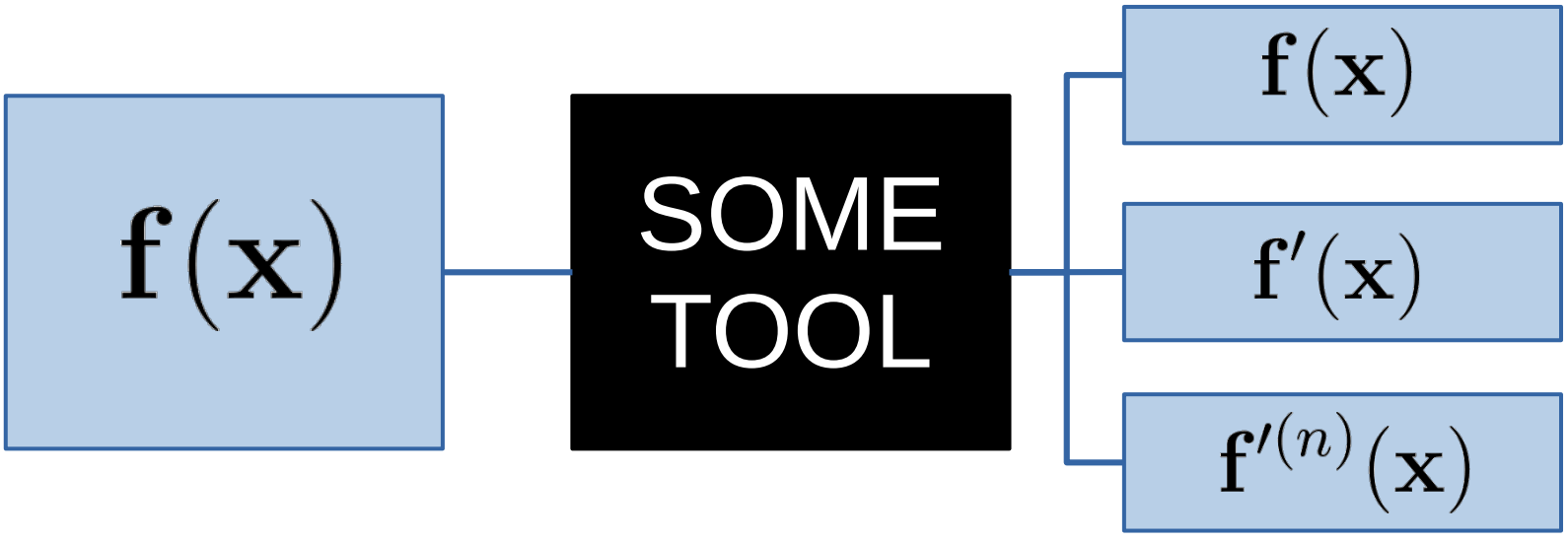
The idea



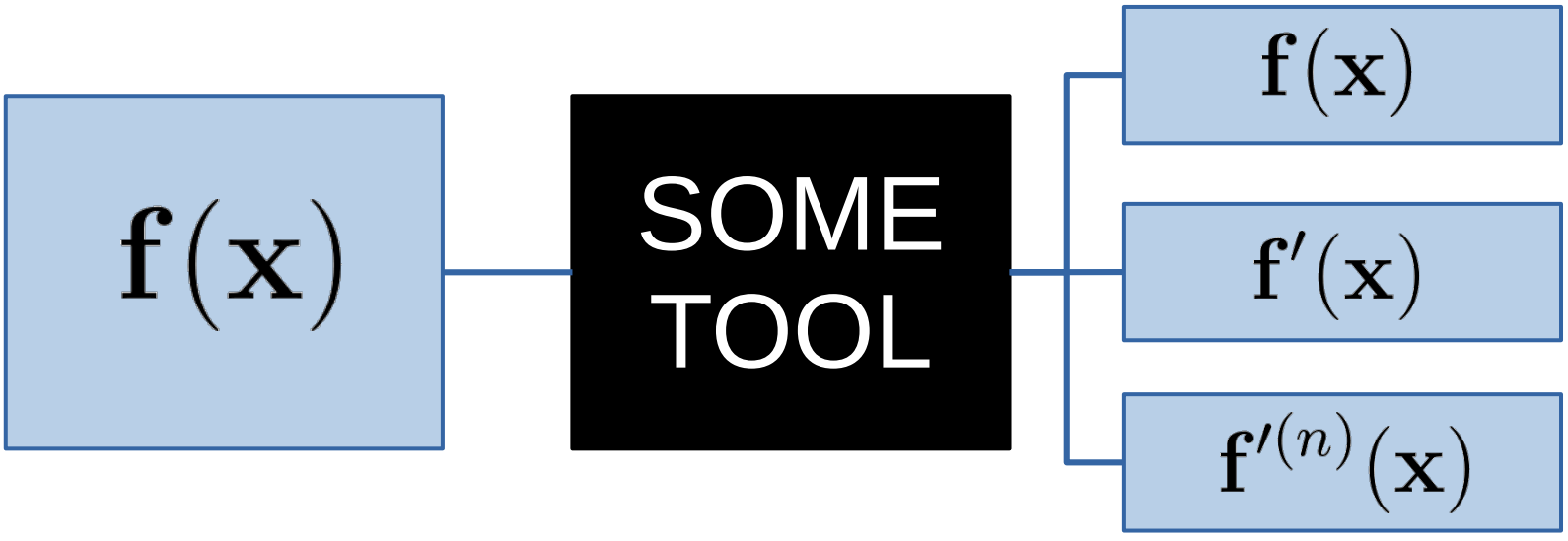
The idea



The idea


$\mathbf{f}(\mathbf{x})$
SOME TOOL
$\mathbf{f}(\mathbf{x})$
$\mathbf{f}'(\mathbf{x})$
$\mathbf{f}'^{(n)}(\mathbf{x})$

The idea



Maybe some complicated nonlinear function

Maybe required by some algorithm (e.g. nonlinear solver, optimisation alg.)

Outline

1. Motivation
2. `for (auto framework : {auto-diff, symb-diff})`
   - Description & explanation
   - Relevant `deal.II` classes
   - Examples
   - Tips
3. Final remarks

Motivation

Rapid prototyping

- Experiment with new models and formulations
- Quicker to implement than hand linearisation
  - For many academic applications, most “research hours” spent in development rather than actual simulation time

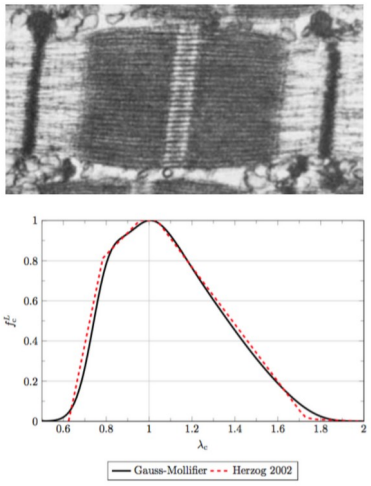
Convenience

- Simple extension of multi-physics code
- Compute higher-order derivatives

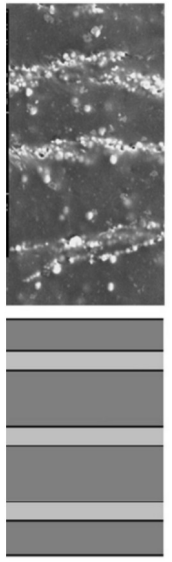
Correctness

- Verify implementations, help establish bugs in linearisation
  - Subtle bugs can have large consequences
  - Unstable formulations (non-convex material models)
- Rule out sources of error
  - “Is my problem’s instability due to an inconsistent linearisation?”

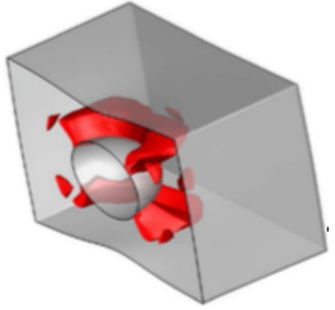
Active skeletal muscle



MREs



EEAPs



... and more...

Soils, concrete, rocks, ceramics, metals, polymers, composites, etc...

Motivation

Nonlinear problems are hard.
There are many failure points when considering highly nonlinear problems.

- Physics & discretisation
  - Appropriate formulation? Appropriate choice of FE basis? Requires stabilisation? Appropriate h-/p- discretisation? Appropriate quadrature rule?
  - Stability conditions (e.g. CFL number); Instabilities (e.g. solution bifurcations)
- Time integration scheme
- Non-linear solution scheme
  - Newton-Raphson or similar; consistent tangent
  - Line search (under-relaxation), arc-length methods (load control)
  - Boundary conditions (maybe non-linear / solution-dependent)
- Constitutive model (nonlinearities, rate-dependence, local variables)
- Linear solver (iterative)
  - Correct solver, adequate preconditioner, appropriate solver settings?

Motivation

Nonlinear problems are hard.
There are many failure points when considering highly nonlinear problems.

- Physics & discretisation
  - Appropriate formulation? Appropriate choice of FE basis? Requires stabilisation? Appropriate h-/p- discretisation? Appropriate quadrature rule?
  - Stability conditions (e.g. CFL number); Instabilities (e.g. solution bifurcations)
- Time integration scheme
- Non-linear solution scheme
  - Newton-Raphson or similar; consistent tangent
  - Line search (under-relaxation), arc-length methods (load control)
  - Boundary conditions (maybe non-linear / solution-dependent)
- Constitutive model (nonlinearities, rate-dependence, local variables)
- Linear solver (iterative)
  - Correct solver, adequate preconditioner, appropriate solver settings?

Differentiation libraries

What is automatic differentiation?

A numerical method that can be used to "automatically" compute the first, and perhaps higher-order, derivatives of function(s) with respect to one or more input variables.

$$\boldsymbol{f}(\boldsymbol{x})|_{\boldsymbol{x}=\bar{\boldsymbol{x}}} \quad \rightarrow \quad \boldsymbol{f}'^{(n)}(\boldsymbol{x})|_{\boldsymbol{x}=\bar{\boldsymbol{x}}}$$

- Numerical technique: Need to know the evaluation point $\vec{\boldsymbol{x}}$ when functions $\boldsymbol{f}$ are computed
- Can be accurate to machine precision
  - Different to numerical differentiation
  - Errors due to Taylor series truncation and step-size

AD/SD module: https://dealii.org/current/doxygen/deal.II/group__auto__symb__diff.html

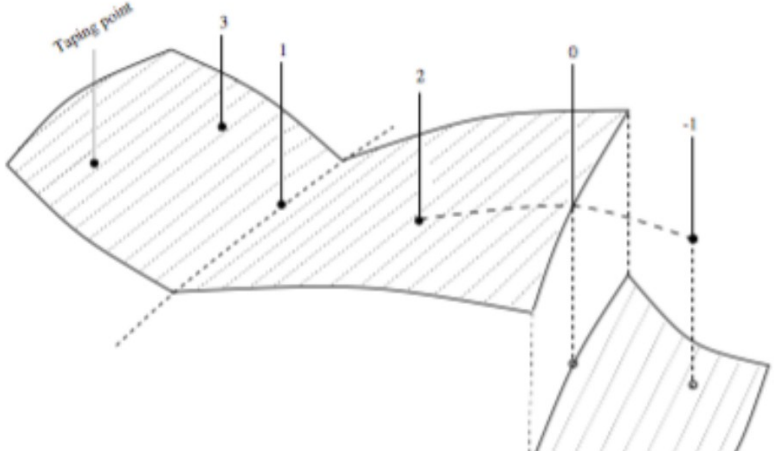
Categorisation of AD frameworks

- Source code transformation (generation)
  - Some code in; new code out
- Operator overloading
  - taping strategies
  - dual / complex-step / hyper-dual numbers (tapeless)

$$f(x+\epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2!}\epsilon^2 f''(x) + \frac{1}{3!}\epsilon^3 f'''(x) + \ldots$$

  - expression templates (compile time operation)
- Forward / Reverse mode

Schematic: ADOL-C taping



Categorisation of AD frameworks

- Source code transformation (generation)
- Operator overloading
- Forward / Reverse mode

$$f(\mathbf{x}) = f_0 \circ f_1 \circ f_2 \circ \ldots \circ f_n(\mathbf{x})$$

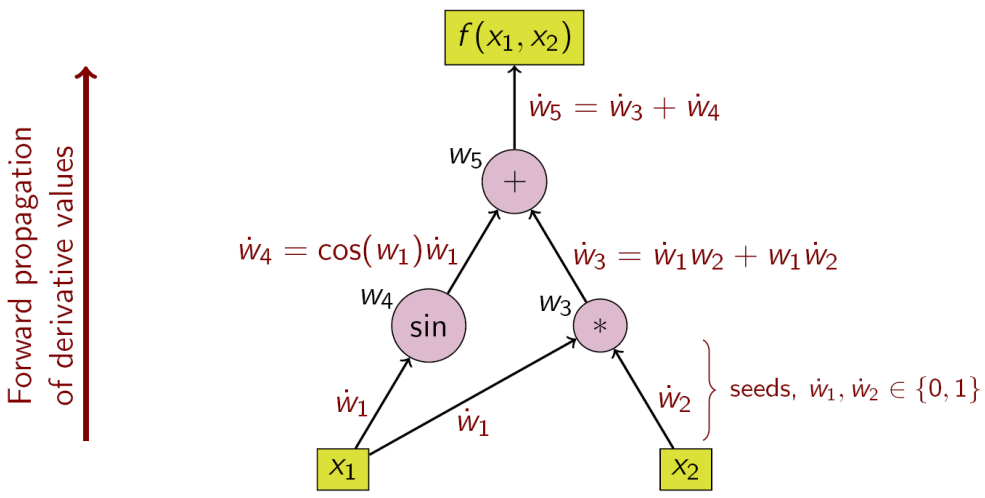
FAD

$$\frac{df(\mathbf{x})}{d\mathbf{x}} = \frac{df_0}{df_1}\left(\frac{df_1}{df_2}\left(\frac{df_2}{df_3}\left(\ldots\left(\frac{df_n(\mathbf{x})}{d\mathbf{x}}\right)\right)\right)\right)$$

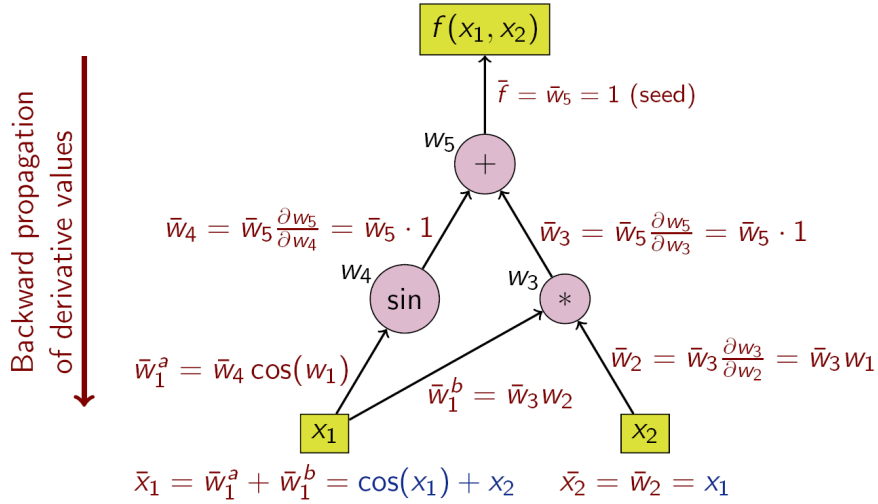
$$\frac{df(\mathbf{x})}{d\mathbf{x}} = \left(\left(\left(\left(\frac{df_0}{df_1}\right)\frac{df_1}{df_2}\right)\frac{df_2}{df_3}\right)\ldots\right)\frac{df_n(\mathbf{x})}{d\mathbf{x}}$$

RAD

FAD: AST & Propogation of derivatives



RAD: AST & Propogation of derivatives



Auto-differentiation helper classes

Differentiation on a finite element (cell-level)

- Residual linearisation (**d** correspond with cell DoFs)

$$\boldsymbol{R}(\boldsymbol{d}) \quad \rightarrow \quad \boldsymbol{K} = \frac{d\boldsymbol{R}(\boldsymbol{d})}{d\boldsymbol{d}}$$

`Differentiation::AD::ResidualLinearization`

- Energy functional (**d** correspond with cell DoFs)

$$\Pi(\boldsymbol{d}) \quad \rightarrow \quad \boldsymbol{R} = \frac{d\,\Pi(\boldsymbol{d})}{d\boldsymbol{d}}, \boldsymbol{K} = \frac{d^2\Pi(\boldsymbol{d})}{d\boldsymbol{d} \otimes d\boldsymbol{d}}$$

`AD::EnergyFunctional`

Differentiation at a evaluation point (quadrature-level)

- Scalar function (accounts for symmetries of **A**)

$$\Psi(\boldsymbol{A}) \quad \rightarrow \quad \frac{d\Psi(\boldsymbol{A})}{d\boldsymbol{A}}, \frac{d^2\Psi(\boldsymbol{A})}{d\boldsymbol{A} \otimes d\boldsymbol{A}}$$

`AD::ScalarFunction`

- Vector function (accounts for symmetries of **A**)

$$\boldsymbol{S}(\boldsymbol{A}) \quad \rightarrow \quad \frac{d\boldsymbol{S}(\boldsymbol{A})}{d\boldsymbol{A}}$$

`AD::VectorFunction`

Example: Residual linearisation

Nonlinear solid elasticity

- Balance of linear momentum excl. inertia (strong form ⟶ weak form + reparameterisation)

$$\nabla \cdot \underbrace{[\mathbf{F} \cdot \mathbf{S}(\mathbf{F})]}_{\mathbf{P}(\mathbf{F})} + \mathbf{b} = \mathbf{0} \quad \text{on } \Omega \quad \longrightarrow \quad (\mathbf{E}', \mathbf{S}(\mathbf{E}))_\Omega = (\mathbf{u}', \mathbf{b})_\Omega + \langle \mathbf{u}', p\mathbf{N} \rangle_{\partial\Omega}$$

- Deformation gradient, Green-Lagrange strain tensor

$$\mathbf{F} = \mathbf{I} + \nabla \mathbf{u} \qquad \mathbf{E} = \frac{1}{2}\left[\mathbf{F}^T \cdot \mathbf{F} - \mathbf{I}\right]$$

- Test function (corresponds with variation of Green-Lagrange strain tensor)

$$\mathbf{E}' = \text{sym}\left[\mathbf{F}^T \cdot \nabla \mathbf{u}'\right]$$

- Residual (continuous) ⟶ Space discretisation ⟶ Residual (discrete)

$$R = \underbrace{(\mathbf{E}', \mathbf{S}(\mathbf{E}))_\Omega}_{R^{\text{int}}} \underbrace{- (\mathbf{u}', \mathbf{b})_\Omega - \langle \mathbf{u}', p\mathbf{N} \rangle_{\partial\Omega}}_{R^{\text{ext}}} \stackrel{!}{=} 0$$

- Jacobian-based nonlinear solvers require linearisation

Example: Residual linearisation

```
1  // Main assembly function
2  // The resulting matrix-based Linear Algebra (LA)
3  // system will be passed off the a linear solver.
4  void assemble_system() {
5
6    // Existing data structures:
7    const FESystem<dim> fe(FE_Q<dim>(...), dim);
8    const FEValuesExtractors::Vector u_fe (...);
9    FEValues<dim> fe_values (...);
10
11   Vector<double> solution (...); // Or another vector type
12   std::vector<types::global_dof_index> local_dof_indices (...);
13   const unsigned int n_q_points (...);
14   FullMatrix<double> cell_matrix (...);
15   Vector<double> cell_rhs (...);
16
```

Example: Residual linearisation

```
1   // Main assembly function
2   // The resulting matrix-based Linear Algebra (LA)
3   // system will be passed off the a linear solver.
4   void assemble_system() {
5
6     // Existing data structures:
7     const FESystem<dim> fe(FE_Q<dim>(...), dim);
8     const FEValuesExtractors::Vector u_fe (...);
9     FEValues<dim> fe_values (...);
10
11    Vector<double> solution (...); // Or another vector type
12    std::vector<types::global_dof_index> local_dof_indices (...);
13    const unsigned int n_q_points (...);
14    FullMatrix<double> cell_matrix (...);
15    Vector<double> cell_rhs (...);
16
```

Assume that the problem is solving for displacement, and needs a vector-valued FE.

Example: Residual linearisation

```
1   // Main assembly function
2   // The resulting matrix-based Linear Algebra (LA)
3   // system will be passed off the a linear solver.
4   void assemble_system() {
5
6     // Existing data structures:
7     const FESystem<dim> fe(FE_Q<dim>(...), dim);
8     const FEValuesExtractors::Vector u_fe (...);
9     FEValues<dim> fe_values (...);
10
11    Vector<double> solution (...); // Or another vector type
12    std::vector<types::global_dof_index> local_dof_indices (...);
13    const unsigned int n_q_points (...);
14    FullMatrix<double> cell_matrix (...);
15    Vector<double> cell_rhs (...);
16
```

Assume that the problem is solving for displacement, and needs a vector-valued FE.

Standard data-structures for matrix-based assembly.

Example: Residual linearisation

```
4   void assemble_system() {
17    // Assembly loop:
18    for (auto &cell : ...)
19    {
20      fe_values.reinit(cell);
21      cell->get_dof_indices(local_dof_indices);
22
23      // Create some aliases for the AD helper.
24      using ADHelper = AD::ResidualLinearization<...>;
25      // e.g. AD::ResidualLinearization<AD::NumberTypes::sacado_dfad, double>
26      // e.g. AD::ResidualLinearization<AD::NumberTypes::adolc_tapeless, double>
27      using ADNumberType = typename ADHelper::ad_type;
28
29      // We'll state the problem in the nomenclature of
30      // the AD framework.
31      const unsigned int n_independent_variables
32        = local_dof_indices.size();
33      const unsigned int n_dependent_variables
34        = local_dof_indices.size();
35
36      // Create and initialize an instance of the helper class.
37      ADHelper ad_helper(n_independent_variables,
38                         n_dependent_variables);
39
40      // Initialize the local data structures for assembly.
41      cell_rhs.reinit(n_dependent_variables);
42      cell_matrix.reinit(n_dependent_variables,
43                         n_independent_variables);
```

Example: Residual linearisation

```
4   void assemble_system() {
17    // Assembly loop:
18    for (auto &cell : ...)
19    {
20      fe_values.reinit(cell);
21      cell->get_dof_indices(local_dof_indices);
22
23      // Create some aliases for the AD helper.
24      using ADHelper = AD::ResidualLinearization<...>;
25      // e.g. AD::ResidualLinearization<AD::NumberTypes::sacado_dfad, double>
26      // e.g. AD::ResidualLinearization<AD::NumberTypes::adolc_tapeless, double>
27      using ADNumberType = typename ADHelper::ad_type;
28
29      // We'll state the problem in the nomenclature of
30      // the AD framework.
31      const unsigned int n_independent_variables
32        = local_dof_indices.size();
33      const unsigned int n_dependent_variables
34        = local_dof_indices.size();
35
36      // Create and initialize an instance of the helper class.
37      ADHelper ad_helper(n_independent_variables,
38                         n_dependent_variables);
39
40      // Initialize the local data structures for assembly.
41      cell_rhs.reinit(n_dependent_variables);
42      cell_matrix.reinit(n_dependent_variables,
43                         n_independent_variables);
```

We'll be working with the local DoFs explicitly, so we need to know then up-front.

Example: Residual linearisation

```
4   void assemble_system() {
17    // Assembly loop:
18    for (auto &cell : ...)
19    {
20      fe_values.reinit(cell);
21      cell->get_dof_indices(local_dof_indices);
22
23      // Create some aliases for the AD helper.
24      using ADHelper = AD::ResidualLinearization<...>;
25      // e.g. AD::ResidualLinearization<AD::NumberTypes::sacado_dfad, double>
26      // e.g. AD::ResidualLinearization<AD::NumberTypes::adolc_tapeless, double>
27      using ADNumberType = typename ADHelper::ad_type;
28
29      // We'll state the problem in the nomenclature of
30      // the AD framework.
31      const unsigned int n_independent_variables
32        = local_dof_indices.size();
33      const unsigned int n_dependent_variables
34        = local_dof_indices.size();
35
36      // Create and initialize an instance of the helper class.
37      ADHelper ad_helper(n_independent_variables,
38                         n_dependent_variables);
39
40      // Initialize the local data structures for assembly.
41      cell_rhs.reinit(n_dependent_variables);
42      cell_matrix.reinit(n_dependent_variables,
43                         n_independent_variables);
```

We'll be working with the local DoFs explicitly, so we need to know then up-front.

Choose the class that will manage AD operations, as well as the framework to be used. The value type is given.

Example: Residual linearisation

```
4   void assemble_system() {
17    // Assembly loop:
18    for (auto &cell : ...)
19    {
20      fe_values.reinit(cell);
21      cell->get_dof_indices(local_dof_indices);
22
23      // Create some aliases for the AD helper.
24      using ADHelper = AD::ResidualLinearization<...>;
25      // e.g. AD::ResidualLinearization<AD::NumberTypes::sacado_dfad, double>
26      // e.g. AD::ResidualLinearization<AD::NumberTypes::adolc_tapeless, double>
27      using ADNumberType = typename ADHelper::ad_type;
28
29      // We'll state the problem in the nomenclature of
30      // the AD framework.
31      const unsigned int n_independent_variables
32        = local_dof_indices.size();
33      const unsigned int n_dependent_variables
34        = local_dof_indices.size();
35
36      // Create and initialize an instance of the helper class.
37      ADHelper ad_helper(n_independent_variables,
38                         n_dependent_variables);
39
40      // Initialize the local data structures for assembly.
41      cell_rhs.reinit(n_dependent_variables);
42      cell_matrix.reinit(n_dependent_variables,
43                         n_independent_variables);
```

We'll be working with the local DoFs explicitly, so we need to know then up-front.

Choose the class that will manage AD operations, as well as the framework to be used. The value type is given.

The standard nomenclature for AD is dependent and independent variables. In general, these need not take the same value.

Example: Residual linearisation

```
4   void assemble_system() {
17    // Assembly loop:
18    for (auto &cell : ...)
19    {
20      fe_values.reinit(cell);
21      cell->get_dof_indices(local_dof_indices);
22
23      // Create some aliases for the AD helper.
24      using ADHelper = AD::ResidualLinearization<...>;
25      // e.g. AD::ResidualLinearization<AD::NumberTypes::sacado_dfad, double>
26      // e.g. AD::ResidualLinearization<AD::NumberTypes::adolc_tapeless, double>
27      using ADNumberType = typename ADHelper::ad_type;
28
29      // We'll state the problem in the nomenclature of
30      // the AD framework.
31      const unsigned int n_independent_variables
32        = local_dof_indices.size();
33      const unsigned int n_dependent_variables
34        = local_dof_indices.size();
35
36      // Create and initialize an instance of the helper class.
37      ADHelper ad_helper(n_independent_variables,
38                         n_dependent_variables);
39
40      // Initialize the local data structures for assembly.
41      cell_rhs.reinit(n_dependent_variables);
42      cell_matrix.reinit(n_dependent_variables,
43                         n_independent_variables);
```

We'll be working with the local DoFs explicitly, so we need to know then up-front.

Choose the class that will manage AD operations, as well as the framework to be used. The value type is given.

The standard nomenclature for AD is dependent and independent variables. In general, these need not take the same value.

Initialise both the helper class, as well as the local system data structures. Note the relation between the index sizes.

Example: Residual linearisation

```
4   void assemble_system() {
46      // First, we set the values for all DoFs.
47      ad_helper.register_dof_values(solution, local_dof_indices);
48
49      // Then we get the complete set of degree of freedom values as
50      // represented by auto-differentiable numbers.
51      // These encode with DoFs correspond with which directional
52      // derivative.
53      const std::vector<ADNumberType> dof_values_ad
54        = ad_helper.get_sensitive_dof_values();
55
56      // Compute all gradients, etc. based on sensitive AD DoF values.
57      std::vector<Tensor<2, dim, ADNumberType>> Grad_u(
58        n_q_points, Tensor<2, dim, ADNumberType>());
59      fe_values[u_fe].get_function_gradients_from_local_dof_values(
60        dof_values_ad, Grad_u);
61
62      // Initialisation
63      std::vector<ADNumberType> residual_ad (
64        n_dependent_variables, ADNumberType(0.0));
```

Example: Residual linearisation

```
4   void assemble_system() {
46      // First, we set the values for all DoFs.
47      ad_helper.register_dof_values(solution, local_dof_indices);
48
49      // Then we get the complete set of degree of freedom values as
50      // represented by auto-differentiable numbers.
51      // These encode with DoFs correspond with which directional
52      // derivative.
53      const std::vector<ADNumberType> dof_values_ad
54        = ad_helper.get_sensitive_dof_values();
55
56      // Compute all gradients, etc. based on sensitive AD DoF values.
57      std::vector<Tensor<2, dim, ADNumberType>> Grad_u(
58        n_q_points, Tensor<2, dim, ADNumberType>());
59      fe_values[u_fe].get_function_gradients_from_local_dof_values(
60        dof_values_ad, Grad_u);
61
62      // Initialisation
63      std::vector<ADNumberType> residual_ad (
64        n_dependent_variables, ADNumberType(0.0));
```

$$\mathbf{R} = \mathbf{R}(\mathbf{u})$$

Extract the working point "**u**" (solution coefficients) for function evaluation. Enumerate the differential components $\mathbf{u}_i$.

Example: Residual linearisation

```
4   void assemble_system() {
46      // First, we set the values for all DoFs.
47      ad_helper.register_dof_values(solution, local_dof_indices);
48
49      // Then we get the complete set of degree of freedom values as
50      // represented by auto-differentiable numbers.
51      // These encode with DoFs correspond with which directional
52      // derivative.
53      const std::vector<ADNumberType> dof_values_ad
54        = ad_helper.get_sensitive_dof_values();
55
56      // Compute all gradients, etc. based on sensitive AD DoF values.
57      std::vector<Tensor<2, dim, ADNumberType>> Grad_u(
58        n_q_points, Tensor<2, dim, ADNumberType>());
59      fe_values[u_fe].get_function_gradients_from_local_dof_values(
60        dof_values_ad, Grad_u);
61
62      // Initialisation
63      std::vector<ADNumberType> residual_ad (
64        n_dependent_variables, ADNumberType(0.0));
```

$$\mathbf{R} = \mathbf{R}(\mathbf{u})$$

Extract the working point "**u**" (solution coefficients) for function evaluation. Enumerate the differential components $\mathbf{u}_i$.

Get the equivalent evaluation point, but in the AD representation that will be tracking the operations performed with them. The solution coefficients are the independent variables "**u**".

Example: Residual linearisation

```
4   void assemble_system() {
46      // First, we set the values for all DoFs.
47      ad_helper.register_dof_values(solution, local_dof_indices);
48
49      // Then we get the complete set of degree of freedom values as
50      // represented by auto-differentiable numbers.
51      // These encode with DoFs correspond with which directional
52      // derivative.
53      const std::vector<ADNumberType> dof_values_ad
54        = ad_helper.get_sensitive_dof_values();
55
56      // Compute all gradients, etc. based on sensitive AD DoF values.
57      std::vector<Tensor<2, dim, ADNumberType>> Grad_u(
58        n_q_points, Tensor<2, dim, ADNumberType>());
59      fe_values[u_fe].get_function_gradients_from_local_dof_values(
60        dof_values_ad, Grad_u);
61
62      // Initialisation
63      std::vector<ADNumberType> residual_ad (
64        n_dependent_variables, ADNumberType(0.0));
```

$$\mathbf{R} = \mathbf{R}(\mathbf{u})$$

Extract the working point "**u**" (solution coefficients) for function evaluation. Enumerate the differential components $\mathbf{u}_I$.

Get the equivalent evaluation point, but in the AD representation that will be tracking the operations performed with them. The solution coefficients are the independent variables "**u**".

$$\mathbf{u}(\mathbf{x}_q) = \sum_I \boldsymbol{\phi}^I(\mathbf{x}_q) u^I$$

Compute the solution [gradient] at all QPs using the FE shape functions and solution coefficients.

Example: Residual linearisation

```
4   void assemble_system() {
46      // First, we set the values for all DoFs.
47      ad_helper.register_dof_values(solution, local_dof_indices);
48
49      // Then we get the complete set of degree of freedom values as
50      // represented by auto-differentiable numbers.
51      // These encode with DoFs correspond with which directional
52      // derivative.
53      const std::vector<ADNumberType> dof_values_ad
54        = ad_helper.get_sensitive_dof_values();
55
56      // Compute all gradients, etc. based on sensitive AD DoF values.
57      std::vector<Tensor<2, dim, ADNumberType>> Grad_u(
58        n_q_points, Tensor<2, dim, ADNumberType>());
59      fe_values[u_fe].get_function_gradients_from_local_dof_values(
60        dof_values_ad, Grad_u);
61
62      // Initialisation
63      std::vector<ADNumberType> residual_ad (
64        n_dependent_variables, ADNumberType(0.0));
```

$$\mathbf{R} = \mathbf{R}(\mathbf{u})$$

Extract the working point "**u**" (solution coefficients) for function evaluation. Enumerate the differential components $\mathbf{u}_I$.

Get the equivalent evaluation point, but in the AD representation that will be tracking the operations performed with them. The solution coefficients are the independent variables "**u**".

$$\mathbf{u}(\mathbf{x}_q) = \sum_I \boldsymbol{\phi}^I(\mathbf{x}_q) u^I$$

Compute the solution [gradient] at all QPs using the FE shape functions and solution coefficients.

Create a data structure to store the dependent variables "**R**".

Example: Residual linearisation

```
4   void assemble_system() {
66      // Compute the cell total residual
67      //   = (internal + external) contributions
68      for (const unsigned int q_point
69           : fe_values.quadrature_point_indices())
70      {
71        // Kinematic variable
72        const Tensor<2, dim, ADNumberType> F =
73          unit_symmetric_tensor<dim>() + Grad_u[q_point];
74
75        // Kinetic variable (potentially deduced from a
76        // non-linear constitutive law)
77        const SymmetricTensor<2,dim,ad_type> S = get_S(F);
78
79        // Add contribution of the internal forces
80        for (const unsigned int I : fe_values.dof_indices())
81        {
82          // Test function. Note the geometric non-linearity.
83          const SymmetricTensor<2,dim,ADNumberType> dE_I
84          = symmetrize(transpose(F)  *
85          fe_values[u_fe].gradient(I,q_point));
86
87          // Accumulate residual contribution to the
88          // selected equation.
89          residual_ad[I] += (dE_I*S) * fe_values.JxW(q_point);
90        }
91      }
92
```

Example: Residual linearisation

```
4   void assemble_system() {
66      // Compute the cell total residual
67      //   = (internal + external) contributions
68      for (const unsigned int q_point
69           : fe_values.quadrature_point_indices())
70      {
71        // Kinematic variable
72        const Tensor<2, dim, ADNumberType> F =
73          unit_symmetric_tensor<dim>() + Grad_u[q_point];
74
75        // Kinetic variable (potentially deduced from a
76        // non-linear constitutive law)
77        const SymmetricTensor<2,dim,ad_type> S = get_S(F);
78
79        // Add contribution of the internal forces
80        for (const unsigned int I : fe_values.dof_indices())
81        {
82          // Test function. Note the geometric non-linearity.
83          const SymmetricTensor<2,dim,ADNumberType> dE_I
84          = symmetrize(transpose(F)  *
85          fe_values[u_fe].gradient(I,q_point));
86
87          // Accumulate residual contribution to the
88          // selected equation.
89          residual_ad[I] += (dE_I*S) * fe_values.JxW(q_point);
90        }
91      }
92
```

$$\mathbf{F} = \mathbf{I} + \nabla \mathbf{u}$$

The local kinematic variables are formed from the solution coefficient.

Example: Residual linearisation

```
4   void assemble_system() {
66      // Compute the cell total residual
67      //   = (internal + external) contributions
68      for (const unsigned int q_point
69           : fe_values.quadrature_point_indices())
70      {
71        // Kinematic variable
72        const Tensor<2, dim, ADNumberType> F =
73          unit_symmetric_tensor<dim>() + Grad_u[q_point];
74
75        // Kinetic variable (potentially deduced from a
76        // non-linear constitutive law)
77        const SymmetricTensor<2,dim,ad_type> S = get_S(F);
78
79        // Add contribution of the internal forces
80        for (const unsigned int I : fe_values.dof_indices())
81        {
82          // Test function. Note the geometric non-linearity.
83          const SymmetricTensor<2,dim,ADNumberType> dE_I
84          = symmetrize(transpose(F)  *
85          fe_values[u_fe].gradient(I,q_point));
86
87          // Accumulate residual contribution to the
88          // selected equation.
89          residual_ad[I] += (dE_I*S) * fe_values.JxW(q_point);
90        }
91      }
92
```

$$\mathbf{F} = \mathbf{I} + \nabla \mathbf{u}$$

The local kinematic variables are formed from the solution coefficient.

A (non-linear) constitutive law relates the kinetic variables to the kinematic ones.

Example: Residual linearisation

```
4   void assemble_system() {
66      // Compute the cell total residual
67      //   = (internal + external) contributions
68      for (const unsigned int q_point
69           : fe_values.quadrature_point_indices())
70      {
71        // Kinematic variable
72        const Tensor<2, dim, ADNumberType> F =
73          unit_symmetric_tensor<dim>() + Grad_u[q_point];
74
75        // Kinetic variable (potentially deduced from a
76        // non-linear constitutive law)
77        const SymmetricTensor<2,dim,ad_type> S = get_S(F);
78
79        // Add contribution of the internal forces
80        for (const unsigned int I : fe_values.dof_indices())
81        {
82          // Test function. Note the geometric non-linearity.
83          const SymmetricTensor<2,dim,ADNumberType> dE_I
84          = symmetrize(transpose(F)  *
85          fe_values[u_fe].gradient(I,q_point));
86
87          // Accumulate residual contribution to the
88          // selected equation.
89          residual_ad[I] += (dE_I*S) * fe_values.JxW(q_point);
90        }
91      }
92
```

$$\mathbf{F} = \mathbf{I} + \nabla \mathbf{u}$$

The local kinematic variables are formed from the solution coefficient.

A (non-linear) constitutive law relates the kinetic variables to the kinematic ones.

$$\mathbf{E}' = \mathrm{sym}\left[\mathbf{F}^T \cdot \nabla \mathbf{u}'\right]$$

Finite-strain elasticity introduces this second, geometric nonlinearity that must be linearised.

Example: Residual linearisation

```
4   void assemble_system() {
66      // Compute the cell total residual
67      //   = (internal + external) contributions
68      for (const unsigned int q_point
69           : fe_values.quadrature_point_indices())
70      {
71        // Kinematic variable
72        const Tensor<2, dim, ADNumberType> F =
73          unit_symmetric_tensor<dim>() + Grad_u[q_point];
74
75        // Kinetic variable (potentially deduced from a
76        // non-linear constitutive law)
77        const SymmetricTensor<2,dim,ad_type> S = get_S(F);
78
79        // Add contribution of the internal forces
80        for (const unsigned int I : fe_values.dof_indices())
81        {
82          // Test function. Note the geometric non-linearity.
83          const SymmetricTensor<2,dim,ADNumberType> dE_I
84          = symmetrize(transpose(F)  *
85          fe_values[u_fe].gradient(I,q_point));
86
87          // Accumulate residual contribution to the
88          // selected equation.
89          residual_ad[I] += (dE_I*S) * fe_values.JxW(q_point);
90        }
91      }
92
```

$$\mathbf{F} = \mathbf{I} + \nabla \mathbf{u}$$

The local kinematic variables are formed from the solution coefficient.

A (non-linear) constitutive law relates the kinetic variables to the kinematic ones.

$$\mathbf{E}' = \mathrm{sym}\left[\mathbf{F}^T \cdot \nabla \mathbf{u}'\right]$$

Finite-strain elasticity introduces this second, geometric nonlinearity that must be linearised.

$$\underbrace{(\mathbf{E}', \mathbf{S}(\mathbf{E}))_\Omega}_{R^{\mathrm{int}}}$$

Example: Residual linearisation

```
 4  void assemble_system() {
93      // External forces: Solution-dependent load?
94      for (const unsigned int face : ...)
95        if (cell->face(face)->at_boundary())
96          residual_ad[I] += ...
97
98      // Finalise
99      ad_helper.register_residual_vector(residual_ad);
100   }
101
102   // Extract the local contributions to the LA system
103   ad_helper.compute_residual(cell_rhs);
104   cell_rhs *= -1.0; // RHS = - residual
105   ad_helper.compute_linearization(cell_matrix);
106
107   // Scatter local system into the global LA system
108   constraints.distribute(...);
109 }
110
111 }
```

Example: Residual linearisation

```
 4  void assemble_system() {
93      // External forces: Solution-dependent load?
94      for (const unsigned int face : ...)
95        if (cell->face(face)->at_boundary())
96          residual_ad[I] += ...
97
98      // Finalise
99      ad_helper.register_residual_vector(residual_ad);
100   }
101
102   // Extract the local contributions to the LA system
103   ad_helper.compute_residual(cell_rhs);
104   cell_rhs *= -1.0; // RHS = - residual
105   ad_helper.compute_linearization(cell_matrix);
106
107   // Scatter local system into the global LA system
108   constraints.distribute(...);
109 }
110
111 }
```

$$\underbrace{-(\mathbf{u}', \mathbf{b})_\Omega - \langle \mathbf{u}', p\mathbf{N} \rangle_{\partial\Omega}}_{R^{\text{ext}}}$$

If the loads are solution dependent, then this can be accommodated as well.

Example: Residual linearisation

```
4   void assemble_system() {
93      // External forces: Solution-dependent load?
94      for (const unsigned int face : ...)
95        if (cell->face(face)->at_boundary())
96          residual_ad[I] += ...
97
98      // Finalise
99      ad_helper.register_residual_vector(residual_ad);
100   }
101
102   // Extract the local contributions to the LA system
103   ad_helper.compute_residual(cell_rhs);
104   cell_rhs *= -1.0; // RHS = - residual
105   ad_helper.compute_linearization(cell_matrix);
106
107   // Scatter local system into the global LA system
108   constraints.distribute(...);
109 }
110
111 }
```

$$\underbrace{-(\mathbf{u}', \mathbf{b})_\Omega - \langle \mathbf{u}', p\mathbf{N} \rangle_{\partial\Omega}}_{R^{\text{ext}}}$$

If the loads are solution dependent, then this can be accommodated as well.

Finalise the accumulated residual contributions.

Example: Residual linearisation

```
4   void assemble_system() {
93      // External forces: Solution-dependent load?
94      for (const unsigned int face : ...)
95        if (cell->face(face)->at_boundary())
96          residual_ad[I] += ...
97
98      // Finalise
99      ad_helper.register_residual_vector(residual_ad);
100   }
101
102   // Extract the local contributions to the LA system
103   ad_helper.compute_residual(cell_rhs);
104   cell_rhs *= -1.0; // RHS = - residual
105   ad_helper.compute_linearization(cell_matrix);
106
107   // Scatter local system into the global LA system
108   constraints.distribute(...);
109 }
110
111 }
```

$$\underbrace{-(\mathbf{u}', \mathbf{b})_\Omega - \langle \mathbf{u}', p\mathbf{N} \rangle_{\partial\Omega}}_{R^{\text{ext}}}$$

If the loads are solution dependent, then this can be accommodated as well.

Finalise the accumulated residual contributions.

Recover the numerical values associated with residual entries $\mathbf{R}_I(\mathbf{u})$

Example: Residual linearisation

```
4   void assemble_system() {
93      // External forces: Solution-dependent load?
94      for (const unsigned int face : ...)
95        if (cell->face(face)->at_boundary())
96          residual_ad[I] += ...
97
98      // Finalise
99      ad_helper.register_residual_vector(residual_ad);
100   }
101
102   // Extract the local contributions to the LA system
103   ad_helper.compute_residual(cell_rhs);
104   cell_rhs *= -1.0; // RHS = - residual
105   ad_helper.compute_linearization(cell_matrix);
106
107   // Scatter local system into the global LA system
108   constraints.distribute(...);
109 }
110
111 }
```

$$\underbrace{-(\mathbf{u}', \mathbf{b})_\Omega - \langle \mathbf{u}', p\mathbf{N} \rangle_{\partial\Omega}}_{R^{\text{ext}}}$$

If the loads are solution dependent, then this can be accommodated as well.

Finalise the accumulated residual contributions.

Recover the numerical values associated with residual entries $\mathbf{R}_I(\mathbf{u})$

Recover all directional derivatives $d\mathbf{R}_I(\mathbf{u})/d\mathbf{u}_J$

Tips for using the AD framework

- Different frameworks have different implementations
  - We currently support Sacado (a Trilinos package) and ADOL-C
  - Performance differs
  - If one doesn’t work, then try another (if possible)
- Forward vs reverse
  - FAD: More dependent variables (outputs) than independants (inputs)
  - RAD: More independent variables (inputs) than dependants (outputs)
- Time derivatives (cf. step-91)
  - Need to account for time integration coefficient during consistent linearisation
    (1) Requires solution and its rate to be treated as independent variables
    (2) Specialised energy functionals (rate-dependant materials)

Tips for using the AD framework

- Sacado
  - Do initialise explicitly with zeros
  - Do not use `auto` to store intermediates

```
using ADNumberType = typename ADHelper::ad_type; // Sacado FAD type
...
const std::vector<ADNumberType> dof_values_ad
  = ad_helper.get_sensitive_dof_values();
std::vector<ADNumberType>> soln(n_q_points);
fe_values[extr].get_function_values_from_local_dof_values(
  dof_values_ad, soln);
...
std::vector<ADNumberType> residual_ad (n_dependent_variables,
                                       ADNumberType(0.0)); // Yes!
...

for (unsigned int I = 0; I < n_dofs_per_cell; ++I) {
  const auto tmp = 5 + soln[I]; // No! :-(
```

  - Consider mixing RAD & FAD when computing HO derivatives

What is symbolic ~~differentiation~~ algebra?

A **Computer Algebra System (CAS)** for manipulating algebraic expressions

$$f(x, y(x, z)) = 3x + y^2 \quad , \quad y = x/z \quad \rightarrow \quad \frac{df}{dx} = 3 + 2y/z$$

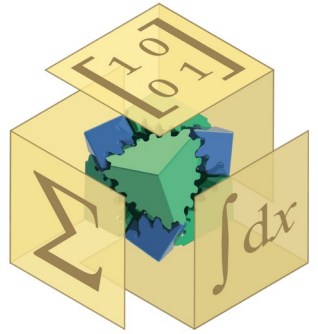
- SD is differentiation of generic expressions having no numerical context
  - Think `Symbol` in SymPy (equivalent functionality in Mathematica, `sym()` in Matlab, etc.)
- CAS not limited to differentiation, but can also perform other mathematical operations (e.g. integration, factorisation, logical operators, etc.)
- Full control over interpretation of symbolic expressions
  - Numerical values
  - Symbolic variables and functions
  - Other constructs...
- “Slow” in native representation (in comparison to AD, for example)
  - We’ll come back to this point later...

AD/SD module: https://dealii.org/current/doxygen/deal.II/group__auto__symb__diff.html

Core symbolic classes & functions

`Differentiation::SD::SymEngineNumber`

An operator-overloaded interface to SymEngine, which is a modern, high-performance CAS.



- Expose functions to:
  - initialise as an expression
  - perform math operations in the natural way (operator and math function overloading)
  - compute derivatives with repect to scalar symbols
  - partial / full substitution
  - evaluation (result conversion to numerical types)
  - ...

Core symbolic classes & functions

Elementary example 1: Boolean expressions

```
1  // Construct symbolic expressions
2  const Expression x("x");
3  const Expression y("y");
4
5  // Construct expressions using operator overloading
6  const Expression f_plus = (x + y)*(x + y);
7  // Parsing expressions from a string is also possible. Its arguments
8  // must have been previously declared through (and in scope).
9  const Expression f_minus ("(x-y)*(x-y)", true);
10
11 // Constructing a conditional expression
12 const Expression f((x > Expression(0.0)), f_plus, f_minus);
13
14 // Value substitution
15 types::substitution_map substitution_map;
16 substitution_map[x] = Expression(1);
17 substitution_map[y] = Expression(2.5);
18 const double evaluated_f =
19   f.substitute_and_evaluate<double>(substitution_map);
20 // Since the substituted value for x was greater than zero, we expect
21 // that the returned result now in evaluated_f was evaluated from
22 // the function f_plus.
23
```

Core symbolic classes & functions

Elementary example 1: Boolean expressions

```
1  // Construct symbolic expressions
2  const Expression x("x");
3  const Expression y("y");
4
5  // Construct expressions using operator overloading
6  const Expression f_plus = (x + y)*(x + y);
7  // Parsing expressions from a string is also possible. Its arguments
8  // must have been previously declared through (and in scope).
9  const Expression f_minus ("(x-y)*(x-y)", true);
10
11 // Constructing a conditional expression
12 const Expression f((x > Expression(0.0)), f_plus, f_minus);
13
14 // Value substitution
15 types::substitution_map substitution_map;
16 substitution_map[x] = Expression(1);
17 substitution_map[y] = Expression(2.5);
18 const double evaluated_f =
19   f.substitute_and_evaluate<double>(substitution_map);
20 // Since the substituted value for x was greater than zero, we expect
21 // that the returned result now in evaluated_f was evaluated from
22 // the function f_plus.
23
```

Arguments to functions.
Use unique strings / names!

Core symbolic classes & functions

Elementary example 1: Boolean expressions

```
1   // Construct symbolic expressions
2   const Expression x("x");
3   const Expression y("y");
4
5   // Construct expressions using operator overloading
6   const Expression f_plus = (x + y)*(x + y);
7   // Parsing expressions from a string is also possible. Its arguments
8   // must have been previously declared through (and in scope).
9   const Expression f_minus ("(x-y)*(x-y)", true);
10
11  // Constructing a conditional expression
12  const Expression f((x > Expression(0.0)), f_plus, f_minus);
13
14  // Value substitution
15  types::substitution_map substitution_map;
16  substitution_map[x] = Expression(1);
17  substitution_map[y] = Expression(2.5);
18  const double evaluated_f =
19    f.substitute_and_evaluate<double>(substitution_map);
20  // Since the substituted value for x was greater than zero, we expect
21  // that the returned result now in evaluated_f was evaluated from
22  // the function f_plus.
23
```

Arguments to functions.
Use unique strings / names!

Expressions can be created by operator and function overloading (i.e. “math”) or by parsing strings.

Core symbolic classes & functions

Elementary example 1: Boolean expressions

```
1  // Construct symbolic expressions
2  const Expression x("x");
3  const Expression y("y");
4
5  // Construct expressions using operator overloading
6  const Expression f_plus = (x + y)*(x + y);
7  // Parsing expressions from a string is also possible. Its arguments
8  // must have been previously declared through (and in scope).
9  const Expression f_minus ("(x-y)*(x-y)", true);
10
11 // Constructing a conditional expression
12 const Expression f((x > Expression(0.0)), f_plus, f_minus);
13
14 // Value substitution
15 types::substitution_map substitution_map;
16 substitution_map[x] = Expression(1);
17 substitution_map[y] = Expression(2.5);
18 const double evaluated_f =
19   f.substitute_and_evaluate<double>(substitution_map);
20 // Since the substituted value for x was greater than zero, we expect
21 // that the returned result now in evaluated_f was evaluated from
22 // the function f_plus.
23
```

Arguments to functions.
Use unique strings / names!

Expressions can be created by operator and function overloading (i.e. “math”) or by parsing strings.

Branching logic can be encoded in expressions.

Core symbolic classes & functions

Elementary example 1: Boolean expressions

```
1  // Construct symbolic expressions
2  const Expression x("x");
3  const Expression y("y");
4
5  // Construct expressions using operator overloading
6  const Expression f_plus = (x + y)*(x + y);
7  // Parsing expressions from a string is also possible. Its arguments
8  // must have been previously declared through (and in scope).
9  const Expression f_minus ("(x-y)*(x-y)", true);
10
11 // Constructing a conditional expression
12 const Expression f((x > Expression(0.0)), f_plus, f_minus);
13
14 // Value substitution
15 types::substitution_map substitution_map;
16 substitution_map[x] = Expression(1);
17 substitution_map[y] = Expression(2.5);
18 const double evaluated_f =
19   f.substitute_and_evaluate<double>(substitution_map);
20 // Since the substituted value for x was greater than zero, we expect
21 // that the returned result now in evaluated_f was evaluated from
22 // the function f_plus.
23
```

Arguments to functions.
Use unique strings / names!

Expressions can be created by operator and function overloading (i.e. “math”) or by parsing strings.

Branching logic can be encoded in expressions.

Substitution maps provide the relation between variables and values (or variables and expressions).

Core symbolic classes & functions

Elementary example 1: Boolean expressions

```
1  // Construct symbolic expressions
2  const Expression x("x");
3  const Expression y("y");
4
5  // Construct expressions using operator overloading
6  const Expression f_plus = (x + y)*(x + y);
7  // Parsing expressions from a string is also possible. Its arguments
8  // must have been previously declared through (and in scope).
9  const Expression f_minus ("(x-y)*(x-y)", true);
10
11 // Constructing a conditional expression
12 const Expression f((x > Expression(0.0)), f_plus, f_minus);
13
14 // Value substitution
15 types::substitution_map substitution_map;
16 substitution_map[x] = Expression(1);
17 substitution_map[y] = Expression(2.5);
18 const double evaluated_f =
19   f.substitute_and_evaluate<double>(substitution_map);
20 // Since the substituted value for x was greater than zero, we expect
21 // that the returned result now in evaluated_f was evaluated from
22 // the function f_plus.
23
```

Arguments to functions.
Use unique strings / names!

Expressions can be created by operator and function overloading (i.e. “math”) or by parsing strings.

Branching logic can be encoded in expressions.

Substitution maps provide the relation between variables and values (or variables and expressions).

“Evaluation” converts a string-like result to a floating point value.

Core symbolic classes & functions

Elementary example 2: Scalar differentiation

```
28  // Construct symbolic expressions
29  const Expression x("x");
30  const Expression f("x**2", true);
31
32  // Now perform differentiation. Specifically, we differentiate the
33  // function "f" with respect to the symbolic variable "x".
34  // The result should be the expression "2*x".
35  const Expression df_dx = f.differentiate(x);
36
37  // Value substitution
38  types::substitution_map substitution_map;
39  substitution_map[x] = Expression(10.0);
40  const double evaluated_df_dx =
41    evaluated_df_dx.substitute_and_evaluate<double>(substitution_map);
42  // We can expect the above to evaluate to "2*10" which is,
43  // of course, the numeric value 20.
```

Core symbolic classes & functions

Elementary example 2: Scalar differentiation

```
28  // Construct symbolic expressions
29  const Expression x("x");
30  const Expression f("x**2", true);
31
32  // Now perform differentiation. Specifically, we differentiate the
33  // function "f" with respect to the symbolic variable "x".
34  // The result should be the expression "2*x".
35  const Expression df_dx = f.differentiate(x);
36
37  // Value substitution
38  types::substitution_map substitution_map;
39  substitution_map[x] = Expression(10.0);
40  const double evaluated_df_dx =
41    evaluated_df_dx.substitute_and_evaluate<double>(substitution_map);
42  // We can expect the above to evaluate to "2*10" which is,
43  // of course, the numeric value 20.
```

Arguments to functions.
Direct creation of an expression.

Core symbolic classes & functions

Elementary example 2: Scalar differentiation

```
28  // Construct symbolic expressions
29  const Expression x("x");
30  const Expression f("x**2", true);
31
32  // Now perform differentiation. Specifically, we differentiate the
33  // function "f" with respect to the symbolic variable "x".
34  // The result should be the expression "2*x".
35  const Expression df_dx = f.differentiate(x);
36
37  // Value substitution
38  types::substitution_map substitution_map;
39  substitution_map[x] = Expression(10.0);
40  const double evaluated_df_dx =
41    evaluated_df_dx.substitute_and_evaluate<double>(substitution_map);
42  // We can expect the above to evaluate to "2*10" which is,
43  // of course, the numeric value 20.
```

Arguments to functions.
Direct creation of an expression.

Scalar differentiation of any expression can be performed with respect to elementary scalar variables.

Core symbolic classes & functions

Elementary example 2: Scalar differentiation

```
28  // Construct symbolic expressions
29  const Expression x("x");
30  const Expression f("x**2", true);
31
32  // Now perform differentiation. Specifically, we differentiate the
33  // function "f" with respect to the symbolic variable "x".
34  // The result should be the expression "2*x".
35  const Expression df_dx = f.differentiate(x);
36
37  // Value substitution
38  types::substitution_map substitution_map;
39  substitution_map[x] = Expression(10.0);
40  const double evaluated_df_dx =
41    evaluated_df_dx.substitute_and_evaluate<double>(substitution_map);
42  // We can expect the above to evaluate to "2*10" which is,
43  // of course, the numeric value 20.
```

Arguments to functions.
Direct creation of an expression.

Scalar differentiation of any expression can be performed with respect to elementary scalar variables.

Choose the evaluation point, and perform the numerical evaluation at that point.

The expression is reusable, and can be evaluated at multiple points.

Core symbolic classes & functions

Convenience functions

- Interaction with symbolic equivalents of commonly used linear algebra classes (scalar types, `Tensor`, `SymmetricTensor`)
  - Scalar & tensor Initialisation

```
template <int rank, int dim>
Tensor<rank, dim, Expression>
make_tensor_of_symbols(const std::string &symbol);
```

  - Scalar & tensor differentiation (arbitrary order, accounts for symmetries)

```
template <int rank_1, int rank_2, int dim>
SymmetricTensor<rank_1 + rank_2, dim, Expression>
differentiate(const SymmetricTensor<rank_1, dim, Expression> &S1,
              const SymmetricTensor<rank_2, dim, Expression> &S2);
```

- Substitution maps
  - Creation (scalar, tensor independent variables)
  - Partial substitution (scalar, tensor dependent variables)
  - Resolution of explicit dependencies

```
types::substitution_map
resolve_explicit_dependencies(const types::substitution_map &substitution_map,
                              const bool force_cyclic_dependency_resolution);
```

Interesting features of a CAS

(Tensor) differentiation

- Delayed resolution of argument parameterisation
  - Partial derivatives

$$\begin{array}{lcl} \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}) & \to & \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B})}{\partial \boldsymbol{A}}\Big|_{\boldsymbol{B}} \\ & d/d\mathbf{A} & \\ \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A})) & \to & \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A}))}{\partial \boldsymbol{A}}\Big|_{\boldsymbol{B}} + \dfrac{\partial \boldsymbol{S}(\boldsymbol{A},\boldsymbol{B}(\boldsymbol{A}))}{\partial \boldsymbol{B}}\Big|_{\boldsymbol{A}} \dfrac{d\boldsymbol{B}(\boldsymbol{A})}{d\boldsymbol{A}} \end{array}$$

Example: Dissipative materials using internal variables

- Power balance ⟶ Dissipation inequality ⟶ Identification of kinetic conjugate quantities

$$\mathscr{D}_{\text{int}} = \mathbf{S}^{\text{tot}} : \frac{1}{2}\dot{\mathbf{C}} - \mathbb{B}\cdot\dot{\mathbb{H}} - \dot{\Psi}_0\left(\mathbf{C}, \mathbf{C}_{\text{v}}^i, \mathbb{H}, \mathbb{H}_{\text{v}}^j\right) \geq 0 \quad \longrightarrow \quad \mathscr{D}_{\text{int}} = \left[\mathbf{S}^{\text{tot}} - 2\frac{\partial \Psi_0}{\partial \mathbf{C}}\right] : \frac{1}{2}\dot{\mathbf{C}} - \sum_i \left[2\frac{\partial \Psi_0}{\partial \mathbf{C}_{\text{v}}^i}\right] : \frac{1}{2}\dot{\mathbf{C}}_{\text{v}}^i$$

$$+ \left[-\mathbb{B} - \frac{\partial \Psi_0}{\partial \mathbb{H}}\right]\cdot\dot{\mathbb{H}} + \sum_j \left[-\frac{\partial \Psi_0}{\partial \mathbb{H}_{\text{v}}^j}\right]\cdot\dot{\mathbb{H}}_{\text{v}}^j \geq 0$$

$$\overline{\mathbf{C}}_{\text{v}}^i = \overline{\mathbf{C}}_{\text{v}}^i\left(\overline{\mathbf{C}}, t\right)$$

Interesting features of a CAS

(Tensor) differentiation

- Delayed resolution of argument parameterisation

Parameter substitution

- Incomplete substitution

$$f(a, b(a)) = a + b \quad \rightarrow \quad f|_{a=1, b=a+2} = 3 + a$$

- Complete substitution

$$f(a(b), b) = a + b \quad \rightarrow \quad f|_{a=b+2, b=1} = [b + 2 + b]_{b=1} = 4$$

Acceleration of symbolic function evaluation

`BatchOptimizer`

A class that transforms symbolic expressions into a different format that can be evaluated much faster than the native (underlying) SymEngine type.

- “Batch” = Simultaneous evaluation of many expressions at once
  - Common subexpression elimination (CSE)
- Expression transformation
  - “lambda”: String to function transformation
  - LLVM: Just in time (JIT) compilation to LLVM byte-code
    - Aggressive optimisation for vectorised expressions
    - Compiler-level CSE

Example: Material law linearisation

```
49  namespace SD = Differentiation::SD;
50
51  template <int dim>
52  class PointHistory
53  {
54  public:
55    PointHistory(const double mu /* Shear modulus */,
56                 const double kappa /* Bulk modulus*/);
57
58    // Update kinetic variables and tangents
59    void
60    update_values(const Tensor<2, dim> &Grad_u_n);
61
62    // Getters for kinetic variables and tangents
63    const SymmetricTensor<2, dim> &
64    get_S() const {return S;}
65
66    const SymmetricTensor<4, dim> &
67    get_H() const {return H;}
68
```

Example: Material law linearisation

```
49  namespace SD = Differentiation::SD;
50
51  template <int dim>
52  class PointHistory
53  {
54  public:
55    PointHistory(const double mu /* Shear modulus */,
56                 const double kappa /* Bulk modulus*/);
57
58    // Update kinetic variables and tangents
59    void
60    update_values(const Tensor<2, dim> &Grad_u_n);
61
62    // Getters for kinetic variables and tangents
63    const SymmetricTensor<2, dim> &
64    get_S() const {return S;}
65
66    const SymmetricTensor<4, dim> &
67    get_H() const {return H;}
68
```

Convenience alias

Example: Material law linearisation

```
49  namespace SD = Differentiation::SD;
50
51  template <int dim>
52  class PointHistory
53  {
54  public:
55    PointHistory(const double mu /* Shear modulus */,
56                 const double kappa /* Bulk modulus*/);
57
58    // Update kinetic variables and tangents
59    void
60    update_values(const Tensor<2, dim> &Grad_u_n);
61
62    // Getters for kinetic variables and tangents
63    const SymmetricTensor<2, dim> &
64    get_S() const {return S;}
65
66    const SymmetricTensor<4, dim> &
67    get_H() const {return H;}
68
```

Convenience alias

Public interface hides the implementation.
(No magic types to be seen here.)

Example: Material law linearisation

```
52  class PointHistory
69  private:
70    // Type definitions
71    using SDSubstitutionMapType = SD::types::substitution_map;
72    using SDNumberType          = SD::Expression;
73    using OptimizerType         = SD::BatchOptimizer<double>;
74
75    // Material parameters
76    const double mu; // Shear modulus
77    const double kappa; // Bulk modulus
78
79    // Symbolic variables (inputs)
80    const SDNumberType mu_SD; // Shear modulus
81    const SDNumberType kappa_SD; // Bulk modulus
82    const SymmetricTensor<2, dim, SDNumberType> C_SD; // RCG Deformation tensor
83
84    // Symbolic outputs
85    SymmetricTensor<2, dim, SDNumberType> S_SD; // Second Piola-Kirchhoff stress
86    SymmetricTensor<4, dim, SDNumberType> H_SD; // Referential tangent
87
88    // Accelerator for evaluation
89    OptimizerType optimizer; // Batch optimizer
90
91    // Evaluated stress and tangent
92    SymmetricTensor<2, dim> S; // Second Piola-Kirchhoff stress
93    SymmetricTensor<4, dim> H; // Referential tangent
94
95    void
96    setup_lqp();
97
98    SDSubstitutionMapType
99    make_substitution_map(const Tensor<2, dim> &Grad_u_n);
100 };
```

Example: Material law linearisation

```
52  class PointHistory
69  private:
70    // Type definitions
71    using SDSubstitutionMapType = SD::types::substitution_map;
72    using SDNumberType          = SD::Expression;
73    using OptimizerType         = SD::BatchOptimizer<double>;
74
75    // Material parameters
76    const double mu; // Shear modulus
77    const double kappa; // Bulk modulus
78
79    // Symbolic variables (inputs)
80    const SDNumberType mu_SD; // Shear modulus
81    const SDNumberType kappa_SD; // Bulk modulus
82    const SymmetricTensor<2, dim, SDNumberType> C_SD; // RCG Deformation tensor
83
84    // Symbolic outputs
85    SymmetricTensor<2, dim, SDNumberType> S_SD; // Second Piola-Kirchhoff stress
86    SymmetricTensor<4, dim, SDNumberType> H_SD; // Referential tangent
87
88    // Accelerator for evaluation
89    OptimizerType optimizer; // Batch optimizer
90
91    // Evaluated stress and tangent
92    SymmetricTensor<2, dim> S; // Second Piola-Kirchhoff stress
93    SymmetricTensor<4, dim> H; // Referential tangent
94
95    void
96    setup_lqp();
97
98    SDSubstitutionMapType
99    make_substitution_map(const Tensor<2, dim> &Grad_u_n);
100 };
```

Some convenience aliases

Example: Material law linearisation

```
52  class PointHistory
69  private:
70    // Type definitions
71    using SDSubstitutionMapType = SD::types::substitution_map;
72    using SDNumberType          = SD::Expression;
73    using OptimizerType         = SD::BatchOptimizer<double>;
74
75    // Material parameters
76    const double mu; // Shear modulus
77    const double kappa; // Bulk modulus
78
79    // Symbolic variables (inputs)
80    const SDNumberType mu_SD; // Shear modulus
81    const SDNumberType kappa_SD; // Bulk modulus
82    const SymmetricTensor<2, dim, SDNumberType> C_SD; // RCG Deformation tensor
83
84    // Symbolic outputs
85    SymmetricTensor<2, dim, SDNumberType> S_SD; // Second Piola-Kirchhoff stress
86    SymmetricTensor<4, dim, SDNumberType> H_SD; // Referential tangent
87
88    // Accelerator for evaluation
89    OptimizerType optimizer; // Batch optimizer
90
91    // Evaluated stress and tangent
92    SymmetricTensor<2, dim> S; // Second Piola-Kirchhoff stress
93    SymmetricTensor<4, dim> H; // Referential tangent
94
95    void
96    setup_lqp();
97
98    SDSubstitutionMapType
99    make_substitution_map(const Tensor<2, dim> &Grad_u_n);
100 };
```

Some convenience aliases

Real valued data for material law

Example: Material law linearisation

```
52  class PointHistory
69  private:
70    // Type definitions
71    using SDSubstitutionMapType = SD::types::substitution_map;
72    using SDNumberType          = SD::Expression;
73    using OptimizerType         = SD::BatchOptimizer<double>;
74
75    // Material parameters
76    const double mu; // Shear modulus
77    const double kappa; // Bulk modulus
78
79    // Symbolic variables (inputs)
80    const SDNumberType mu_SD; // Shear modulus
81    const SDNumberType kappa_SD; // Bulk modulus
82    const SymmetricTensor<2, dim, SDNumberType> C_SD; // RCG Deformation tensor
83
84    // Symbolic outputs
85    SymmetricTensor<2, dim, SDNumberType> S_SD; // Second Piola-Kirchhoff stress
86    SymmetricTensor<4, dim, SDNumberType> H_SD; // Referential tangent
87
88    // Accelerator for evaluation
89    OptimizerType optimizer; // Batch optimizer
90
91    // Evaluated stress and tangent
92    SymmetricTensor<2, dim> S; // Second Piola-Kirchhoff stress
93    SymmetricTensor<4, dim> H; // Referential tangent
94
95    void
96    setup_lqp();
97
98    SDSubstitutionMapType
99    make_substitution_map(const Tensor<2, dim> &Grad_u_n);
100 };
```

Some convenience aliases

Real valued data for material law

Symbolic data as input to material law
- Constitutive parameters
- Parameterisation

Example: Material law linearisation

```
52  class PointHistory
69  private:
70    // Type definitions
71    using SDSubstitutionMapType = SD::types::substitution_map;
72    using SDNumberType          = SD::Expression;
73    using OptimizerType         = SD::BatchOptimizer<double>;
74
75    // Material parameters
76    const double mu; // Shear modulus
77    const double kappa; // Bulk modulus
78
79    // Symbolic variables (inputs)
80    const SDNumberType mu_SD; // Shear modulus
81    const SDNumberType kappa_SD; // Bulk modulus
82    const SymmetricTensor<2, dim, SDNumberType> C_SD; // RCG Deformation tensor
83
84    // Symbolic outputs
85    SymmetricTensor<2, dim, SDNumberType> S_SD; // Second Piola-Kirchhoff stress
86    SymmetricTensor<4, dim, SDNumberType> H_SD; // Referential tangent
87
88    // Accelerator for evaluation
89    OptimizerType optimizer; // Batch optimizer
90
91    // Evaluated stress and tangent
92    SymmetricTensor<2, dim> S; // Second Piola-Kirchhoff stress
93    SymmetricTensor<4, dim> H; // Referential tangent
94
95    void
96    setup_lqp();
97
98    SDSubstitutionMapType
99    make_substitution_map(const Tensor<2, dim> &Grad_u_n);
100 };
```

Some convenience aliases

Real valued data for material law

Symbolic data as input to material law
- Constitutive parameters
- Parameterisation

Symbolic representation of stresses and linearisation.

Fast evaluator of symbolic expressions.

Example: Material law linearisation

```
52  class PointHistory
69  private:
70    // Type definitions
71    using SDSubstitutionMapType = SD::types::substitution_map;
72    using SDNumberType          = SD::Expression;
73    using OptimizerType         = SD::BatchOptimizer<double>;
74
75    // Material parameters
76    const double mu; // Shear modulus
77    const double kappa; // Bulk modulus
78
79    // Symbolic variables (inputs)
80    const SDNumberType mu_SD; // Shear modulus
81    const SDNumberType kappa_SD; // Bulk modulus
82    const SymmetricTensor<2, dim, SDNumberType> C_SD; // RCG Deformation tensor
83
84    // Symbolic outputs
85    SymmetricTensor<2, dim, SDNumberType> S_SD; // Second Piola-Kirchhoff stress
86    SymmetricTensor<4, dim, SDNumberType> H_SD; // Referential tangent
87
88    // Accelerator for evaluation
89    OptimizerType optimizer; // Batch optimizer
90
91    // Evaluated stress and tangent
92    SymmetricTensor<2, dim> S; // Second Piola-Kirchhoff stress
93    SymmetricTensor<4, dim> H; // Referential tangent
94
95    void
96    setup_lqp();
97
98    SDSubstitutionMapType
99    make_substitution_map(const Tensor<2, dim> &Grad_u_n);
100 };
```

Some convenience aliases

Real valued data for material law

Symbolic data as input to material law
- Constitutive parameters
- Parameterisation

Symbolic representation of stresses and linearisation.

Fast evaluator of symbolic expressions.

Real valued stresses and linearisation, evaluated at material point.

Example: Material law linearisation

```
52  class PointHistory
69  private:
70    // Type definitions
71    using SDSubstitutionMapType = SD::types::substitution_map;
72    using SDNumberType          = SD::Expression;
73    using OptimizerType         = SD::BatchOptimizer<double>;
74
75    // Material parameters
76    const double mu; // Shear modulus
77    const double kappa; // Bulk modulus
78
79    // Symbolic variables (inputs)
80    const SDNumberType mu_SD; // Shear modulus
81    const SDNumberType kappa_SD; // Bulk modulus
82    const SymmetricTensor<2, dim, SDNumberType> C_SD; // RCG Deformation tensor
83
84    // Symbolic outputs
85    SymmetricTensor<2, dim, SDNumberType> S_SD; // Second Piola-Kirchhoff stress
86    SymmetricTensor<4, dim, SDNumberType> H_SD; // Referential tangent
87
88    // Accelerator for evaluation
89    OptimizerType optimizer; // Batch optimizer
90
91    // Evaluated stress and tangent
92    SymmetricTensor<2, dim> S; // Second Piola-Kirchhoff stress
93    SymmetricTensor<4, dim> H; // Referential tangent
94
95    void
96    setup_lqp();
97
98    SDSubstitutionMapType
99    make_substitution_map(const Tensor<2, dim> &Grad_u_n);
100 };
```

Some convenience aliases

Real valued data for material law

Symbolic data as input to material law
- Constitutive parameters
- Parameterisation

Symbolic representation of stresses and linearisation.

Fast evaluator of symbolic expressions.

Real valued stresses and linearisation, evaluated at material point.

Initialisation functions

Example: Material law linearisation

```
103  template <int dim>
104  PointHistory<dim>::
105  PointHistory(const double mu,
106               const double kappa)
107    : mu(mu)
108    , kappa(kappa)
109    , mu_SD("mu")
110    , kappa_SD("kappa")
111    , C_SD(SD::make_symmetric_tensor_of_symbols<2, dim>("C"))
112    , optimizer(SD::OptimizerType::llvm,
113                SD::OptimizationFlags::optimize_all)
114  {
115    setup_symbol_differentiation();
116  }
```

Example: Material law linearisation

```
103  template <int dim>
104  PointHistory<dim>::
105  PointHistory(const double mu,
106               const double kappa)
107    : mu(mu)
108    , kappa(kappa)
109    , mu_SD("mu")
110    , kappa_SD("kappa")
111    , C_SD(SD::make_symmetric_tensor_of_symbols<2, dim>("C"))
112    , optimizer(SD::OptimizerType::llvm,
113                SD::OptimizationFlags::optimize_all)
114  {
115    setup_symbol_differentiation();
116  }
```

Create unique symbolic variables
- Scalar
- Symmetric tensor

Example: Material law linearisation

```
103  template <int dim>
104  PointHistory<dim>::
105  PointHistory(const double mu,
106               const double kappa)
107    : mu(mu)
108    , kappa(kappa)
109    , mu_SD("mu")
110    , kappa_SD("kappa")
111    , C_SD(SD::make_symmetric_tensor_of_symbols<2, dim>("C"))
112    , optimizer(SD::OptimizerType::llvm,
113                SD::OptimizationFlags::optimize_all)
114  {
115    setup_symbol_differentiation();
116  }
```

Create unique symbolic variables
- Scalar
- Symmetric tensor

Initialise batch optimiser: It will compile the constructed expressions and aggressively optimise them.

Example: Material law linearisation

```
103  template <int dim>
104  PointHistory<dim>::
105  PointHistory(const double mu,
106               const double kappa)
107    : mu(mu)
108    , kappa(kappa)
109    , mu_SD("mu")
110    , kappa_SD("kappa")
111    , C_SD(SD::make_symmetric_tensor_of_symbols<2, dim>("C"))
112    , optimizer(SD::OptimizerType::llvm,
113                SD::OptimizationFlags::optimize_all)
114  {
115    setup_symbol_differentiation();
116  }
```

Create unique symbolic variables
- Scalar
- Symmetric tensor

Initialise batch optimiser: It will compile the constructed expressions and aggressively optimise them.

Construct and finalise all symbolic expressions once up-front.

Example: Material law linearisation

```
119  template <int dim>
120  void
121  PointHistory<dim>::
122  setup_symbol_differentiation()
123  {
124    // Step 0: Define some symbols (done in the PointHistory constructor)
125    // Step 1: Update stress and material tangent
126    {
127      // Perform some symbolic calculations
128      // - Kinematic quantities
129      const SDNumberType det_C_SD = determinant(C_SD);
130      const SDNumberType det_F_SD = sqrt(det_C_SD);
131      const SymmetricTensor<2, dim, SDNumberType> C_bar_SD(
132        pow(det_C_SD, -1.0 / dim) * C_SD);
133
134      // - Energy function (Could be made generic)
135      const SDNumberType symbolic_psi_iso
136        = mu_SD / 2.0 * (trace(C_bar) - dim);
137
138      const SDNumberType symbolic_psi_vol
139        = (kappa / 4.0) *
140          (det_F_SD * det_F_SD - 1.0 - 2.0 * log(det_F_SD));
141
142      const SDNumberType symbolic_psi = symbolic_psi_vol + symbolic_psi_iso;
143
144      // - Partial derivatives of energy function
145      //   S = 2*dpsi_dC
146      //   H = 2*dS_dC = 4*d2psi_dC_dC
147      symbolic_S = 2.0 * SD::differentiate(symbolic_psi, C_SD);
148      symbolic_H = 2.0 * SD::differentiate(symbolic_S, C_SD);
149    }
```

Example: Material law linearisation

```
119  template <int dim>
120  void
121  PointHistory<dim>::
122  setup_symbol_differentiation()
123  {
124    // Step 0: Define some symbols (done in the PointHistory constructor)
125    // Step 1: Update stress and material tangent
126    {
127      // Perform some symbolic calculations
128      // - Kinematic quantities
129      const SDNumberType det_C_SD = determinant(C_SD);
130      const SDNumberType det_F_SD = sqrt(det_C_SD);
131      const SymmetricTensor<2, dim, SDNumberType> C_bar_SD(
132        pow(det_C_SD, -1.0 / dim) * C_SD);
133
134      // - Energy function (Could be made generic)
135      const SDNumberType symbolic_psi_iso
136        = mu_SD / 2.0 * (trace(C_bar) - dim);
137
138      const SDNumberType symbolic_psi_vol
139        = (kappa / 4.0) *
140          (det_F_SD * det_F_SD - 1.0 - 2.0 * log(det_F_SD));
141
142      const SDNumberType symbolic_psi = symbolic_psi_vol + symbolic_psi_iso;
143
144      // - Partial derivatives of energy function
145      //   S = 2*dpsi_dC
146      //   H = 2*dS_dC = 4*d2psi_dC_dC
147      symbolic_S = 2.0 * SD::differentiate(symbolic_psi, C_SD);
148      symbolic_H = 2.0 * SD::differentiate(symbolic_S, C_SD);
149    }
```

Define some deformation tensors and invariants in a symbolic form. Note the support for tensor functions and standard math language.

Example: Material law linearisation

```
119  template <int dim>
120  void
121  PointHistory<dim>::
122  setup_symbol_differentiation()
123  {
124    // Step 0: Define some symbols (done in the PointHistory constructor)
125    // Step 1: Update stress and material tangent
126    {
127      // Perform some symbolic calculations
128      // - Kinematic quantities
129      const SDNumberType det_C_SD = determinant(C_SD);
130      const SDNumberType det_F_SD = sqrt(det_C_SD);
131      const SymmetricTensor<2, dim, SDNumberType> C_bar_SD(
132        pow(det_C_SD, -1.0 / dim) * C_SD);
133
134      // - Energy function (Could be made generic)
135      const SDNumberType symbolic_psi_iso
136        = mu_SD / 2.0 * (trace(C_bar) - dim);
137
138      const SDNumberType symbolic_psi_vol
139        = (kappa / 4.0) *
140          (det_F_SD * det_F_SD - 1.0 - 2.0 * log(det_F_SD));
141
142      const SDNumberType symbolic_psi = symbolic_psi_vol + symbolic_psi_iso;
143
144      // - Partial derivatives of energy function
145      //   S = 2*dpsi_dC
146      //   H = 2*dS_dC = 4*d2psi_dC_dC
147      symbolic_S = 2.0 * SD::differentiate(symbolic_psi, C_SD);
148      symbolic_H = 2.0 * SD::differentiate(symbolic_S, C_SD);
149    }
```

Define some deformation tensors and invariants in a symbolic form. Note the support for tensor functions and standard math language.

$$\psi_0(\mathbf{C}) = \frac{\mu}{2}\left[\mathrm{tr}(\bar{\mathbf{C}}) - \mathbf{I} : \mathbf{I}\right] + \frac{\kappa}{4}\left[J^2 - 1 - 2\ln(J)\right]$$

Fully symbolic form of (strain) energy function. This defines the constitutive law, and can be reused with different material parameters.

Example: Material law linearisation

```
119  template <int dim>
120  void
121  PointHistory<dim>::
122  setup_symbol_differentiation()
123  {
124    // Step 0: Define some symbols (done in the PointHistory constructor)
125    // Step 1: Update stress and material tangent
126    {
127      // Perform some symbolic calculations
128      // - Kinematic quantities
129      const SDNumberType det_C_SD = determinant(C_SD);
130      const SDNumberType det_F_SD = sqrt(det_C_SD);
131      const SymmetricTensor<2, dim, SDNumberType> C_bar_SD(
132        pow(det_C_SD, -1.0 / dim) * C_SD);
133
134      // - Energy function (Could be made generic)
135      const SDNumberType symbolic_psi_iso
136        = mu_SD / 2.0 * (trace(C_bar) - dim);
137
138      const SDNumberType symbolic_psi_vol
139        = (kappa / 4.0) *
140          (det_F_SD * det_F_SD - 1.0 - 2.0 * log(det_F_SD));
141
142      const SDNumberType symbolic_psi = symbolic_psi_vol + symbolic_psi_iso;
143
144      // - Partial derivatives of energy function
145      //   S = 2*dpsi_dC
146      //   H = 2*dS_dC = 4*d2psi_dC_dC
147      symbolic_S = 2.0 * SD::differentiate(symbolic_psi, C_SD);
148      symbolic_H = 2.0 * SD::differentiate(symbolic_S, C_SD);
149    }
```

Define some deformation tensors and invariants in a symbolic form. Note the support for tensor functions and standard math language.

$$\psi_0(\mathbf{C}) = \frac{\mu}{2}\left[\mathrm{tr}(\bar{\mathbf{C}}) - \mathbf{I} : \mathbf{I}\right] + \frac{\kappa}{4}\left[J^2 - 1 - 2\ln(J)\right]$$

Fully symbolic form of (strain) energy function. This defines the constitutive law, and can be reused with different material parameters.

The stress is computed by differentiaing the energy. Its linearisation is computed by differentiating the stress. Both results are tensorial.

Example: Material law linearisation

```
122   setup_symbol_differentiation()
151     // Step 2: Configure the batch optimizer
152     const SDSubstitutionMapType sub_vals =
153       make_substitution_map(Tensor<2, dim>()); // Values don't matter here
154     optimizer.register_symbols(sub_vals);
155
156     optimizer.register_functions(symbolic_S, symbolic_H);
157
158     optimizer.optimize();
159   }
```

Example: Material law linearisation

```
122   setup_symbol_differentiation()
151     // Step 2: Configure the batch optimizer
152     const SDSubstitutionMapType sub_vals =
153       make_substitution_map(Tensor<2, dim>()); // Values don't matter here
154     optimizer.register_symbols(sub_vals);
155
156     optimizer.register_functions(symbolic_S, symbolic_H);
157
158     optimizer.optimize();
159   }
```

Register the independent variables.

Example: Material law linearisation

```
122   setup_symbol_differentiation()
151     // Step 2: Configure the batch optimizer
152     const SDSubstitutionMapType sub_vals =
153       make_substitution_map(Tensor<2, dim>()); // Values don't matter here
154     optimizer.register_symbols(sub_vals);
155
156     optimizer.register_functions(symbolic_S, symbolic_H);
157
158     optimizer.optimize();
159   }
```

Register the independent variables.

Register the dependent variables.

Example: Material law linearisation

```
122   setup_symbol_differentiation()
151     // Step 2: Configure the batch optimizer
152     const SDSubstitutionMapType sub_vals =
153       make_substitution_map(Tensor<2, dim>()); // Values don't matter here
154     optimizer.register_symbols(sub_vals);
155
156     optimizer.register_functions(symbolic_S, symbolic_H);
157
158     optimizer.optimize();
159   }
```

Register the independent variables.

Register the dependent variables.

Finalise the optimiser (compile expressions if using LLVM).

Example: Material law linearisation

```
160  template <int dim>
161  SDSubstitutionMapType
162  PointHistory<dim>::
163  make_substitution_map(const Tensor<2, dim> &Grad_u_n)
164  {
165    // Compute kinematic variables from inputs
166    const Tensor<2, dim> F =
167      (Tensor<2, dim>(StandardTensors<dim>::I) + Grad_u_n);
168    const SymmetricTensor<2, dim> C = symmetrize(transpose(F) * F);
169
170    // Build the symbol substitution map
171    SDSubstitutionMapType sub_vals_unresolved;
172    // - Add material coefficients
173    SD::add_to_substitution_map(sub_vals_unresolved,
174                                SD::make_substitution_map(mu_SD,
175                                                          mu));
176    SD::add_to_substitution_map(sub_vals_unresolved,
177                                SD::make_substitution_map(kappa_SD,
178                                                          kappa));
179    // - Add kinematic variables
180    SD::add_to_substitution_map(sub_vals_unresolved,
181                                SD::make_substitution_map(C_SD, C));
182
183    // NOTE: The recursive substitution is not really required in this case,
184    // but good to use in practise in case a more complex energy function is
185    // employed later.
186    return SD::resolve_explicit_dependencies(sub_vals_unresolved);
187  }
```

Example: Material law linearisation

```
160  template <int dim>
161  SDSubstitutionMapType
162  PointHistory<dim>::
163  make_substitution_map(const Tensor<2, dim> &Grad_u_n)
164  {
165    // Compute kinematic variables from inputs
166    const Tensor<2, dim> F =
167      (Tensor<2, dim>(StandardTensors<dim>::I) + Grad_u_n);
168    const SymmetricTensor<2, dim> C = symmetrize(transpose(F) * F);
169
170    // Build the symbol substitution map
171    SDSubstitutionMapType sub_vals_unresolved;
172    // - Add material coefficients
173    SD::add_to_substitution_map(sub_vals_unresolved,
174                                SD::make_substitution_map(mu_SD,
175                                                          mu));
176    SD::add_to_substitution_map(sub_vals_unresolved,
177                                SD::make_substitution_map(kappa_SD,
178                                                          kappa));
179    // - Add kinematic variables
180    SD::add_to_substitution_map(sub_vals_unresolved,
181                                SD::make_substitution_map(C_SD, C));
182
183    // NOTE: The recursive substitution is not really required in this case,
184    // but good to use in practise in case a more complex energy function is
185    // employed later.
186    return SD::resolve_explicit_dependencies(sub_vals_unresolved);
187  }
```

Compute values of kinematic fields (from FE solution).

Example: Material law linearisation

```
160  template <int dim>
161  SDSubstitutionMapType
162  PointHistory<dim>::
163  make_substitution_map(const Tensor<2, dim> &Grad_u_n)
164  {
165    // Compute kinematic variables from inputs
166    const Tensor<2, dim> F =
167      (Tensor<2, dim>(StandardTensors<dim>::I) + Grad_u_n);
168    const SymmetricTensor<2, dim> C = symmetrize(transpose(F) * F);
169
170    // Build the symbol substitution map
171    SDSubstitutionMapType sub_vals_unresolved;
172    // - Add material coefficients
173    SD::add_to_substitution_map(sub_vals_unresolved,
174                                SD::make_substitution_map(mu_SD,
175                                                          mu));
176    SD::add_to_substitution_map(sub_vals_unresolved,
177                                SD::make_substitution_map(kappa_SD,
178                                                          kappa));
179    // - Add kinematic variables
180    SD::add_to_substitution_map(sub_vals_unresolved,
181                                SD::make_substitution_map(C_SD, C));
182
183    // NOTE: The recursive substitution is not really required in this case,
184    // but good to use in practise in case a more complex energy function is
185    // employed later.
186    return SD::resolve_explicit_dependencies(sub_vals_unresolved);
187  }
```

Compute values of kinematic fields (from FE solution).

Append entries to substitution map, that maps symbols to values.

Example: Material law linearisation

```
160  template <int dim>
161  SDSubstitutionMapType
162  PointHistory<dim>::
163  make_substitution_map(const Tensor<2, dim> &Grad_u_n)
164  {
165    // Compute kinematic variables from inputs
166    const Tensor<2, dim> F =
167      (Tensor<2, dim>(StandardTensors<dim>::I) + Grad_u_n);
168    const SymmetricTensor<2, dim> C = symmetrize(transpose(F) * F);
169
170    // Build the symbol substitution map
171    SDSubstitutionMapType sub_vals_unresolved;
172    // - Add material coefficients
173    SD::add_to_substitution_map(sub_vals_unresolved,
174                                SD::make_substitution_map(mu_SD,
175                                                          mu));
176    SD::add_to_substitution_map(sub_vals_unresolved,
177                                SD::make_substitution_map(kappa_SD,
178                                                          kappa));
179    // - Add kinematic variables
180    SD::add_to_substitution_map(sub_vals_unresolved,
181                                SD::make_substitution_map(C_SD, C));
182
183    // NOTE: The recursive substitution is not really required in this case,
184    // but good to use in practise in case a more complex energy function is
185    // employed later.
186    return SD::resolve_explicit_dependencies(sub_vals_unresolved);
187  }
```

Compute values of kinematic fields (from FE solution).

Append entries to substitution map, that maps symbols to values.

Ensure that all values in the map are numeric, and not symbolic due to unresolved chains of depdendence.

Example: Material law linearisation

```
190  template <int dim>
191  void
192  PointHistory<dim>::
193  update_values(const Tensor<2, dim> &Grad_u_n)
194  {
195    // Build the symbol substitution map
196    const SDSubstitutionMapType sub_vals =
197      make_substitution_map(Grad_u_n);
198
199    // Pass substitution map to the optimizer
200    optimizer.substitute(sub_vals);
201
202    // Update stress and material tangent
203    // (Compute real-valued deal.II tensors)
204    S = optimizer.evaluate(symbolic_S);
205    H = optimizer.evaluate(symbolic_H);
206  }
```

Example: Material law linearisation

```
190  template <int dim>
191  void
192  PointHistory<dim>::
193  update_values(const Tensor<2, dim> &Grad_u_n)
194  {
195    // Build the symbol substitution map
196    const SDSubstitutionMapType sub_vals =
197      make_substitution_map(Grad_u_n);
198
199    // Pass substitution map to the optimizer
200    optimizer.substitute(sub_vals);
201
202    // Update stress and material tangent
203    // (Compute real-valued deal.II tensors)
204    S = optimizer.evaluate(symbolic_S);
205    H = optimizer.evaluate(symbolic_H);
206  }
```

Construct the substitution map with field values, material coefficients, and any other arguments for the symbolic functions to be evaluated.

Example: Material law linearisation

```
190  template <int dim>
191  void
192  PointHistory<dim>::
193  update_values(const Tensor<2, dim> &Grad_u_n)
194  {
195    // Build the symbol substitution map
196    const SDSubstitutionMapType sub_vals =
197      make_substitution_map(Grad_u_n);
198
199    // Pass substitution map to the optimizer
200    optimizer.substitute(sub_vals);
201
202    // Update stress and material tangent
203    // (Compute real-valued deal.II tensors)
204    S = optimizer.evaluate(symbolic_S);
205    H = optimizer.evaluate(symbolic_H);
206  }
```

Construct the substitution map with field values, material coefficients, and any other arguments for the symbolic functions to be evaluated.

Execute the optimiser, which computes the output values from the inputs passed to it.

Example: Material law linearisation

```
190  template <int dim>
191  void
192  PointHistory<dim>::
193  update_values(const Tensor<2, dim> &Grad_u_n)
194  {
195    // Build the symbol substitution map
196    const SDSubstitutionMapType sub_vals =
197      make_substitution_map(Grad_u_n);
198
199    // Pass substitution map to the optimizer
200    optimizer.substitute(sub_vals);
201
202    // Update stress and material tangent
203    // (Compute real-valued deal.II tensors)
204    S = optimizer.evaluate(symbolic_S);
205    H = optimizer.evaluate(symbolic_H);
206  }
```

Construct the substitution map with field values, material coefficients, and any other arguments for the symbolic functions to be evaluated.

Execute the optimiser, which computes the output values from the inputs passed to it.

Extract components from all of the functions that have been evaluated by the optimiser.

Tips for using the SD framework

- Do use unique names for variables! There are limited checks for variable name clashes.
- For non-trivial functions, use the `BatchOptimizer` to increase performance
  - Leverage CSE: Try to collect as many functions as possible to be simultaneously evaluated. (No advantage to using SymEngine native CSE when using LLVM)
  - Try to set up and persistently store `BatchOptimizer` up-front; reuse many times
- `SymbolicFunction`: Run-time function definition and evaluation (similar to FunctionParser)
  - Interfaces with parameter files
  - Computes gradients, Laplacians, Hessians, and time derivatives
- Some functionality might not exist in SymEngine (e.g. full complex number support)
  - Consider contributing extensions to that library
- Do not try to do symbolic assembly (finite-element level symbolic expression of residual, energy functional)
  - Tried and tested; is possible but VERY computationally expensive
  - Form-type language is better suited this description of the problem

Final remarks

Why would you want to do this?

Leveraging AD/SD reduces the number of equations to be implemented

- Finite element assembly & constitutive modelling

Enhances ability to rapid prototype

- May require a “paradigm shift” (SD)

Which approach should I take?

Choosing a suitable AD library / implementation is not necessarily trivial

Non-negligible performance loss when using AD or SD (from experimentation)

- FEM: Most efficient AD: Cell-level residual linearisation (ADOL-C tapeless)
- FEM: Most efficient SD: Quadrature-point description of constitutive law (with LLVM optimisation)
- Performance penality may be amortised as number of processes increased;
  assembly vs solver time

Easiest framework to use: AD (almost drop-in replacement; with some caveats)

Most robust & flexible framework: Symbolic / SD

Final remarks

What about high-performance computing?

There is some evidence to suggest that these tools could be used along with matrix-free by pre-computing and caching quadrature-level data.
A matrix-free approach for finite-strain hyperelastic problems using geometric multigrid, Davydov et al. (2020), https://arxiv.org/abs/1904.13131

Where can I get more information?

Tutorials: step-71, step-72, [step-73, step-91]

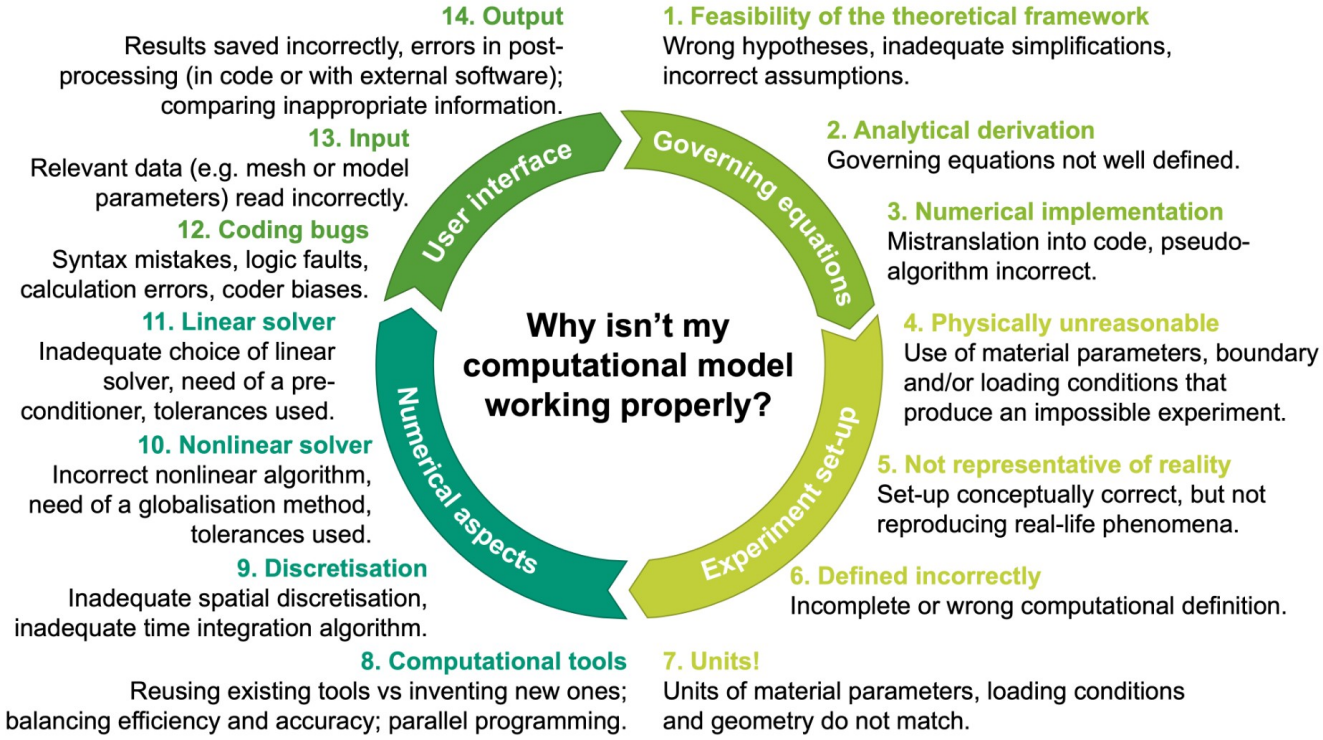
Check out the tests! These include permutations of some tutorials (`step-xx-...`).

- `deal.ii/tests/ad_common_tests`
  - `deal.ii/tests/adolc`
  - `deal.ii/tests/sacado`
- `deal.ii/tests/symengine`

Final remarks

More tips for debugging (and development in general)

I'm stuck! Debugging computational solid mechanics models, Comellas et al. (2022)
https://arxiv.org/abs/2209.04198



Thank you for your attention!

https://dealii.org