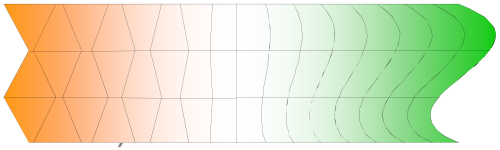
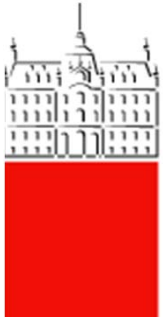
Symbolic Methods and AceGen

Jože Korelc

University of Ljubljana, Slovenia

Mercator Visiting Professor at Leibniz Universität Hannover

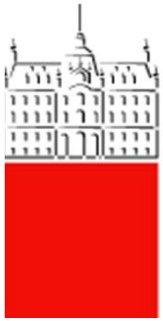




Computational Solution Environments

- General problem solving environment (PSE)
  - general solvers for ODE-s or PDE-s
  - user templates are provided for certain class of problems - ELLPACK, DIFPACK, SCIRun, FlexPDE
  - numerical libraries with compiled functions - NAG
  - interactive numerical environments - MATLAB, FEMLAB
  - symbolic systems Mathematica, Maple
- Object oriented environments
  - collection of objects, clases, methods
  - DIFPACK, FEMTheory
- Specialized finite element enviroments
  - ABAQUS, FEAP, ANSYS, ...

- **Hybrid approaches**

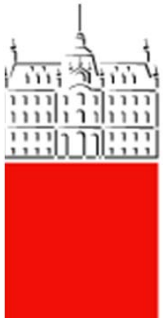


Hybrid approaches

- Steering application provides interfaces to the tools
  - Alice, SCIRun
  - popular for multi-physics, multi-field

- Hybrid object-oriented approach
  - domain-specific language
  - built-in C++ libraries for symbolic manipulation and AD
  - FeniCS, FreeFem++

- Hybrid symbolic–numeric approach
  - general computer algebra system for code generation
  - general finite element environment
  - AceGen, AceFEM

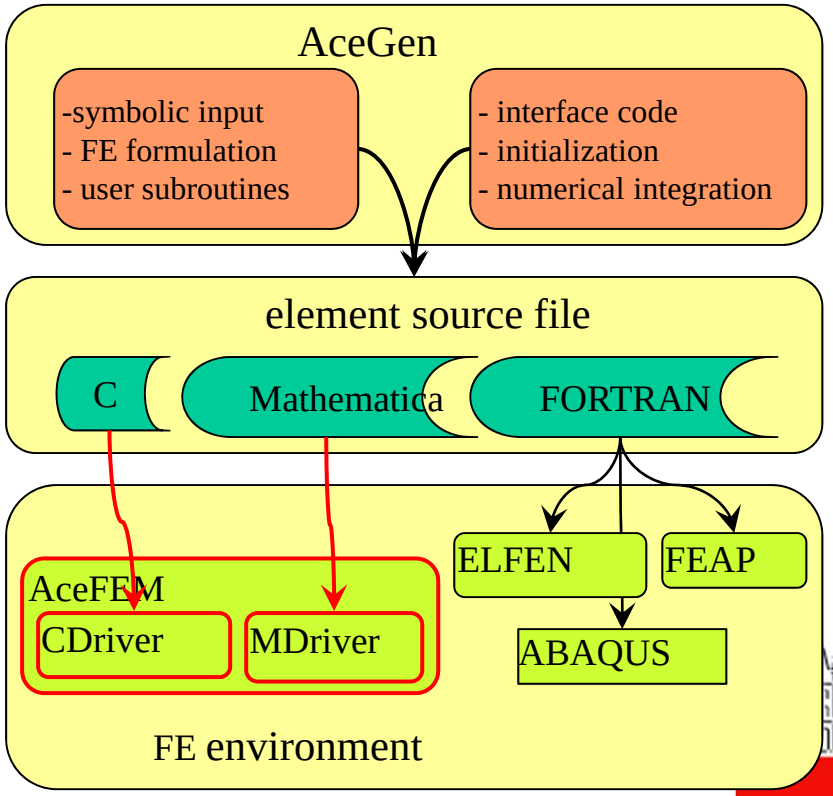


Automatic code generation - AceGen

Multi-language, Multi-environment Numerical Code Generation
www.fgg.uni-lj.si/symech/

Key features:

- simultaneous derivation of expressions
- automatic selection of the appropriate intermediate variables
- forward and backward mode of automatic differentiation technique
- multi-language code generation (Fortran/Fortran90, C, C++,C#, Mathematica© language, Matlab© language),
- automatic interface to FEM environments



Multi-language code generation

1. Mathematical description

$$u = \sum_{i=1}^{3} \hat{N}_i \, \hat{u}_i$$

$$\hat{\mathbf{N}} = \{\frac{x}{L}, 1 - \frac{x}{L}, \frac{x}{L}(1 - \frac{x}{L})\}$$

$$f = u^2$$

$$\nabla f = ?$$

```
<< AceGen`;
SMSInitialize["filename",
  "Language" -> "Mathematica"
SMSModule["Gradf", Real[u$$[3], x$$, L$$, g$$[3]]];
{x, L} ⊢ {SMSReal[x$$], SMSReal[L$$]};
uh ⊢ SMSReal[Table[u$$[i], {i, 3}]];
Nh ⊨ {x/L, 1 - x/L, x/L (1 - x/L)};
u ⊨ Nh.uh;
f ⊨ u^2;
g ⊨ SMSD[f, ui];
SMSExport[g, g$$];
...te[];
```

3. Automatically generated code

"Fortran"

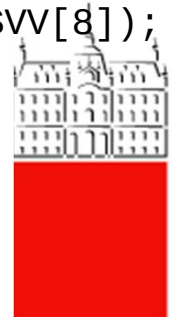
```
SUBROUTINE Test(v,u1,x,L,g)
REAL*8 v(500),u1(3),x,L,g(3)
v(6)=x/L
v(7)=1d0 - v(6)
v(8)=v(6)*v(7)
v(12)=2d0*(u(1)*v(6) +
- u(2)*v(7) + u(3)*v(8))
g(1)=v(12)*v(6)
g(2)=v(12)*v(7)
g(3)=v(12)*v(8)
END
```

"C++"

```
void Test(double v[501],
double u1[3],double *x,
double *L,double g[3])
{
v[6]=*x/*L;
v[7]=1e0 - v[6];
v[8]=v[6]*v[7];
v[12]=2e0*(u[0]*v[6] +
 u[1]*v[7]+ u[2]*v[8]);
g[0]=v[12]*v[6];
g[1]=v[12]*v[7];
g[2]=v[12]*v[8];
};
```

"Mathematica"

```
Test[]:=Module[{},
$VV[6]=x$$/L$$;
$VV[7]=1 - $VV[6];
$VV[8]=$VV[6]*$VV[7];
$VV[12]=2*(u$$[1]*$VV[6] +
  u$$[2]*$VV[7]+u$$[3]*$VV[8]);
g$$[1]=$VV[6]*$VV[12];
g$$[2]=$VV[7]*$VV[12];
g$$[3]=$VV[8]*$VV[12];
]
```



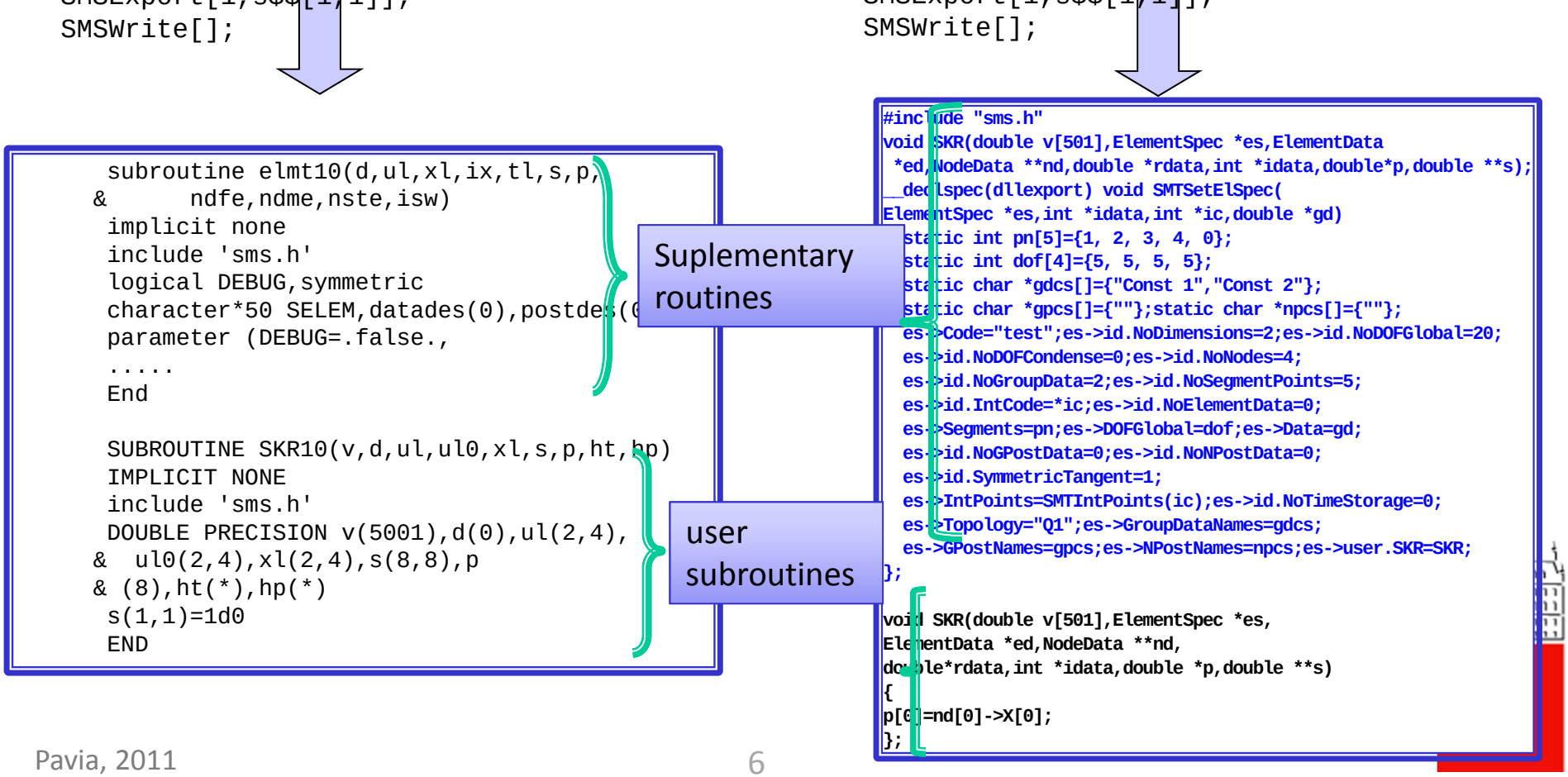
Multi-environment code generation

FEAP

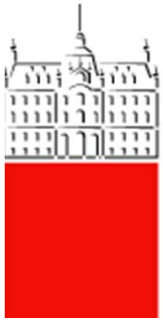
```
 <<AceGen`;
SMSInitialize["test",
  "Environment"→"FEAP"];
SMSTemplate["SMSTopology"→"Q1"];
SMSStandardModule["Tangent and
residual"];
SMSExport[1,s$$[1,1]];
SMSWrite[];
```

AceFEM

```
 <<AceGen`;
SMSInitialize["test",
  "Environment"→"AceFEM"];
SMSTemplate["SMSTopology"→"Q1"];
SMSStandardModule["Tangent and
residual"];
SMSExport[1,s$$[1,1]];
SMSWrite[];
```



Expression optimization



Simultaneous optimization of the expressions

- expressions are optimize immediately after they are derived
- special procedures are needed for non-local operations
- appropriate for large problems where also intermediate results can be subjected to the uncontrolled swell of expressions

Vector of 7 new auxiliary variables:

$$v = \left\{ E, I, L, \frac{12 v_1 v_2}{{v_3}^3}, -\frac{6 v_1 v_2}{{v_3}^2}, \frac{4 v_1 v_2}{v_3}, \frac{v_6}{2} \right\}$$

Original matrix

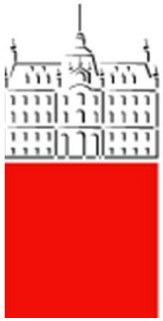
$$\mathbf{K}_0 = \begin{bmatrix} \frac{12EI}{L^3} & -\frac{6EI}{L^2} & -\frac{12EI}{L^3} & -\frac{6EI}{L^2} \\ -\frac{6EI}{L^2} & \frac{4EI}{L} & \frac{6EI}{L^2} & \frac{2EI}{L} \\ -\frac{12EI}{L^3} & \frac{6EI}{L^2} & \frac{12EI}{L^3} & \frac{6EI}{L^2} \\ -\frac{6EI}{L^2} & \frac{2EI}{L} & \frac{6EI}{L^2} & \frac{4EI}{L} \end{bmatrix}$$

AceGen

$$\mathbf{K}_0 = \begin{bmatrix} v_4 & v_5 & -v_4 & v_5 \\ v_5 & v_6 & -v_5 & v_7 \\ -v_4 & -v_5 & v_4 & -v_5 \\ v_5 & v_7 & -v_5 & v_6 \end{bmatrix}$$

Optimized matrix

Result is optimized matrix, expressed with the new auxiliary variables



Automatic theorem proving

How to prove arbitrary (mathematical) statement automatically ?

$$\mathbf{K} \bullet \mathbf{K}^{-1} = \mathbf{I} \quad \checkmark$$

$$\mathbf{K} \bullet \mathbf{K}^{-1} = 2\mathbf{K} \quad //$$

Automatic theorem proving is essential when large expressions derived by a symbolic system need to be simplified.

General symbolic matrices A and B:

```
A = {{a11, a12}, {a21, a22}};

(*symmetric*)

B = {{b11, b12}, {b12, b22}};

(*Quadratic form*)

K = Transpose[A].B.A;
```

```
K[[1, 2]]
```

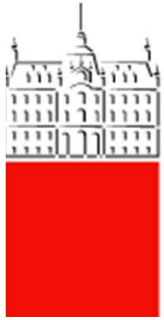
```
a12 (a11 b11 + a21 b12) +
 a22 (a11 b12 + a21 b22)
```

```
K[[2, 1]]
```

```
a11 (a12 b11 + a22 b12) +
 a21 (a12 b12 + a22 b22)
```

```
K[[1, 2]] === K[[2, 1]]
```

```
False
```



Deterministic methods

- general solution is not available
- statements that can be expressed as a system of algebraic equations
  - **statement is correct if the corresponding system of algebraic equations has a solution**
  - **Grobner bases**
    - technique provides algorithmic solution to a system of algebraic (polynomial) equations
    - Grobner bases are computed by the Buchberger´s algorithm
    - **Buchberger´s algorithm** is equivalent to Gauss elimination for the system of polynomial equations
- Mathematica

```
Expand[K[[1, 2]]] === Expand[K[[2, 1]]]
```

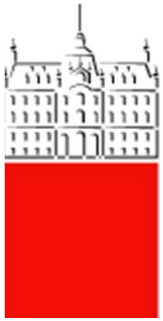
```
True
```

```
Simplify[K[[1, 2]] - K[[2, 1]]]
```

```
0
```

```
PolynomialReduce[K[[1, 2]], {K[[2, 1]]},]
```

```
{{1}, 0}
```



Heuristic methods

- Numerical identification of relations between expressions
  - the simplest way of automatic theorem proving
  - the problem of equivalence testing is determining whether two expressions (different in appearance) are indeed mathematically the same.

$$f(x) = 2x^5 - 2 \neq 0$$

$$f(x) = x^{10} - 1 + (1 - x^5)(x^5 + 1) = 0\,?$$

- The correctness can be numerically determined only with a certain degree of probability

```
x^10 - 1 + (1 - x^5) (x^5 + 1) // Expand
```

```
0
```

```
x^10 - 1 + (1 - x^5) (x^5 + 1) /. x → RandomReal[]
```

$$-1.11022 \times 10^{-16}$$

Signature functions

Numerical identification of relations between set of expressions

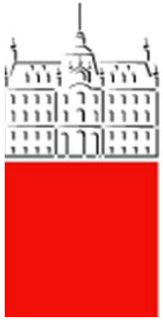
$$
\begin{aligned}
&0 : x \\
&1 : y_1 = f_1(x) \\
&2 : y_2 = f_2(x, y_1) \\
&\ldots. \\
&n : y_n = f_n(x, y_1, \ldots, y_{n-1})
\end{aligned}
\qquad
\begin{aligned}
&0 : S_0 = S(x) \;\; \ldots \;\; random\, signature \\
&1 : S_1 = S(S_0) \\
&2 : S_2 = S(S_0, S_1) \\
&\ldots. \\
&n : S_n = S(S_1, S_2, S_{n-1})
\end{aligned}
$$

$$
S_k - S_j = \begin{cases} \neq 0 & \Rightarrow \quad y_k \neq y_j \\ = 0 & \Rightarrow \quad y_k \equiv y_j \end{cases}
$$

with a probability that can be tuned

Possible signature functions:

1. every expression is mapped into an integer of arbitrary length
   - complex implementation
   - general method
2. every expression is evaluated with the set of real random numbers of arbitrary precission



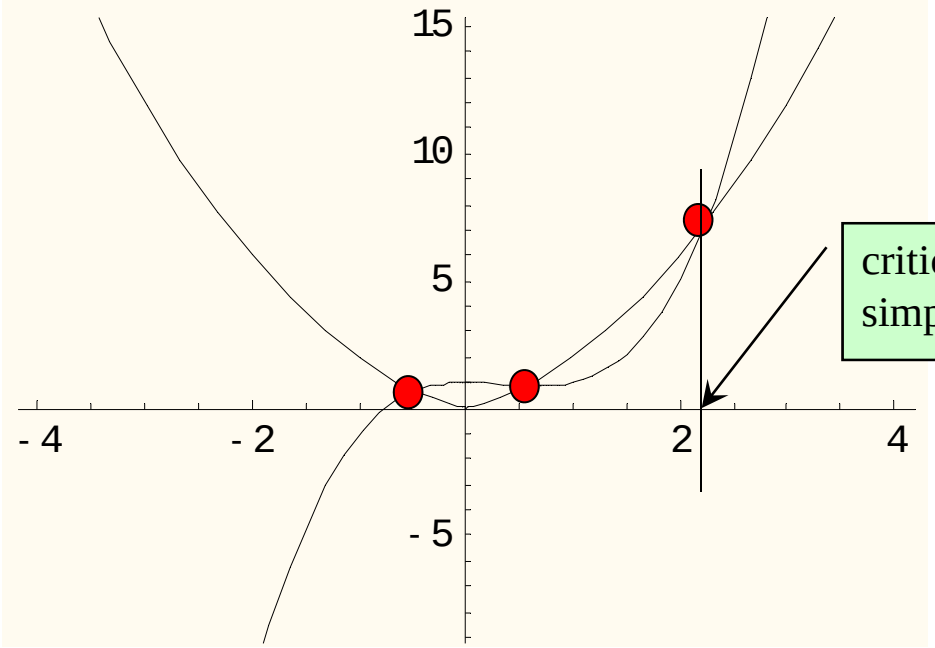
Heuristic code optimization

Example 1:

$f_1 = x^3 - x^2 + 1$

$f_2 = |x| + x^2$

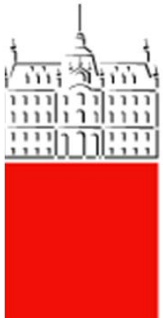
$x \in [-4,4]$



Probability of wrong simplification is for this case is defined by:

- probability that polynomial equation has roots on interval
- floating point precision

Auxiliary variables



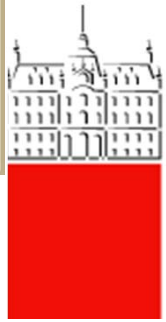
Assignment operators

General algorithm

```
<< AceGen`;
SMSInitialize["test",
  "Language" -> "Mathematica"];
SMSModule["test", Real[x0$$, r$$]];
x0 ⊢ SMSReal[x0$$];
x ⫤ x0;
SMSDo[
  f ⊨ x^2 + 2 Sin[x^3];
  dx ⊨ -f/SMSD[f, x];
  x ⊣ x + dx;
  SMSIf[Abs[dx] < 10^-10,
   SMSBreak[];];
  SMSIf[i == 15, SMSReturn[];
  ];
, {i, 1, 30, 1, {x}}];
SMSExport[x, r$$];
SMSWrite[];
```

Four additional assignment operators in *AceGen* replace the standard Mathematica assignment operator lhs=rhs

lhs⊨*rhs*	Evaluates and optimizes *rhs* and assigns the result to be the value of *lhs*. New auxiliary variable is introduced if needed.
lhs⊢*rhs*	A new auxiliary variable is introduced regardless on contents of *rhs*.
lhs ⫤ *rhs1*	Evaluates and optimizes *rhs1* and assigns the result to be the value of *lhs*. The *lhs* variable can appear after the initialization more than once on a left–hand side of equation.
lhs ⊣ *rhs2*	A new value *rhs2* is assigned to the previously created variable *lhs*.



Conditional statements

"If" construct

$$y = \begin{cases} x^2 & x \geq 0 \\ Sin(x) & x < 0 \end{cases}$$

Mathematica input

```
y = If[x ≥ 0
  , x^2
  , Sin[x]
 ]
```

In-cell AceGen input

```
y ⊨ SMSIf[x ≥ 0
  , x^2
  , Sin[x]
 ]
```

Cross-cell AceGen input

```
SMSIf[x ≥ 0]
y ⫤ x^2;
SMSElse[];
y ⫣ Sin[x];
SMSEndIf[y];
```

auxiliary variable $_1$y ($V[i,1])

auxiliary variable $_2$y ($V[i,2])

auxiliary variable $_3$y ($V[i,3])

Loops

- "Do" construct

$$y = \sum_{i=1}^{n} x^i$$

Mathematica input

```
y = 0; n = 10;
Do[
  y = y + x^i;
  , {i, 1, n, 1}
 ];
```

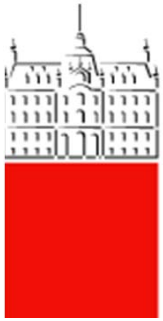
In-cell AceGen input

```
y ↦ 0;
SMSDo[
  y ↤ y + x^i;
  , {i, 1, n, 1, y}
 ];
```

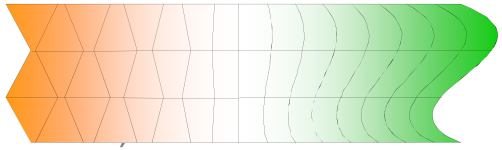
Cross-cell AceGen input

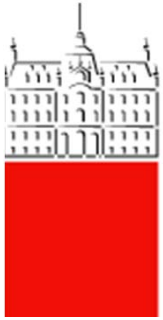
```
y ↦ 0;
SMSDo[ i, 1, n, 1, y];
y ↤ y + x^i;
SMSEndDo[y]
```

$_1y$ $_2y$ $_3y$ $_4y$



Automatic Differentiation





Differentiation

symbolic differentiation

- differentiation of isolated formulas with the symbolic manipulation system or manually
- exact (chain rule)

$$\frac{d(\bullet)}{d(\bullet)} \quad \frac{\partial(\bullet)}{\partial(\bullet)} \quad \delta(\bullet)$$

numerical differentiation

- finite difference approximation
- approximation

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\Delta x, y) - f(x-\Delta x, y)}{2\Delta x}$$

automatic differentiation (AD) => "computational derivative"

- differentiation of the algorithm by the use of chain rule
- exact (except for the round off errors)
- let define "computational derivative" with the following formalism

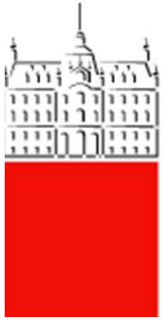
$$\frac{\hat{\delta}(\bullet)}{\hat{\delta}(\bullet)}$$

computational derivative

Automatic differentiation

- **Automatic differentiation technique (AD):**
  - differentiation of the whole program
  - automatic differentiation tool generates a program code for the derivative from a program code for the basic function
  - control structures (If, Do,..) are left unchanged

- **Two basic methods of AD:**
  - forward mode of AD
    - standard "chain rule" on a level of the algorithm
  - backward mode of AD
    - also named: reverse mode, inverse method, adjoint code construction, adjoint sensitivity, ...



Forward versus backward mode

$\mathbf{x}-$ vector of n independent variables $\quad y(\mathbf{x})-$ dependent variable $\quad \nabla y=?$

forward mode

$$
\begin{array}{ll}
1 & b := \sum_{l=1}^{n} x_l^2 \\
2 & c := Sin(b) \\
3 & y := b\,c
\end{array}
$$

$$
\begin{aligned}
\nabla b &= \left\{\frac{db}{dx_l}\right\} = \left\{2\,x_l\,; l = 1,2,\ldots,n\right\} && n \\
\nabla c &= \left\{\frac{dc}{dx_l}\right\} = \left\{Cos(b)\,\nabla b_l; l = 1,2,\ldots,n\right\} && n \\
\nabla y &= \left\{\frac{dy}{dx_l}\right\} = \left\{\nabla b_l\,c + b\,\nabla c_l; l = 1,2,\ldots,n\right\} && n
\end{aligned}
$$

backward mode

reversal of the flow of the program

$\overline{y},\overline{c},\overline{b}$ - adjoint values

$$
\begin{array}{ll}
1 & y := b\,c \\
2 & c := Sin(b) \\
3 & b := \sum_{l=1}^{n} x_l^2
\end{array}
$$

$$
\begin{aligned}
\overline{y} &= \frac{dy}{dy} = 1 && 1 \\
\overline{c} &= \frac{dy}{dc} = \frac{\partial y}{\partial c}\overline{y} = b\,\overline{y} && 1 \\
\overline{b} &= \frac{dy}{db} = \frac{\partial y}{\partial b}\overline{y} + \frac{\partial c}{\partial b}\overline{c} = c\overline{y} + Cos(b)\,\overline{c} && 1 \\
\nabla y &= \left\{\frac{dy}{dx_l}\right\} = \left\{\overline{x}_l\right\} = \left\{\frac{\partial b}{\partial x_l}\,\overline{b}\right\} = \left\{2\,x_l\,\overline{b}; l = 1,2,\ldots,n\right\} && n
\end{aligned}
$$

Efficiency of automatic differentiation

- $wratio(f)$ - work ratio

$$wratio(f) = \frac{\text{cost}(f, \nabla f)}{\text{cost}(f)}$$

- $cost(f)$ - the number of arithmetic operations for evaluating *f*
- $cost(f, \nabla f)$ - the number of arithmetic operations for evaluating *f* and it´s gradient with *n* components

forward mode

- Computational cost proportional to *n* $wratio(f) \approx \alpha(n+1)$

backward mode

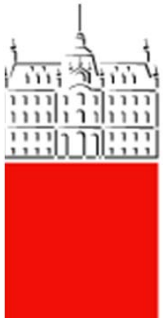
- "If care is taken in handling quantities which are common to the function and derivative, the cost ratio is usually around 1.5, not n+1" (Wolfe, 1982)
- formal proof by Baur and Strassen (1983) $wratio(f) < 5$

Implementation of AD

- Source-to-source translator
  - original source is transformed into derivative code
  - compile-time solution
  - ADIFOR, Odyssee, TAMC
  - minimal changes in original code (declaration of input and output variables)

- Operator overloading
  - modern compilers accept user-defined data types and operator overloading
  - run-time solution
  - low numerical efficiency
  - ADOL-C

$$a + b \quad \Rightarrow \quad \nabla(a + b)$$



Automatic differentiation in AceGen

AceGen enhancements with respect to the standard AD technique:

- AD procedure can be initiated at any time and at any point and as many times as required within the same user subroutine
- AD as code-to-code translator consistently extends current code rather than produce a new one
- the results of all previous uses of AD have to be accounted for when AD is used several times
- mechanism to include exceptions within the AD procedure easily

Interaction AD and expression optimization

Interaction of automatic differentiation and simultaneous optimisation of expressions my lead to wrong results!

Original	*Simplification A*	*Simplification B*
$x := L(a)$	$x := L(a)$	$v_1 := L(a)$
$y := L(a) + x^2$	$y := x + x^2$	$x := v_1$
$\frac{dy}{dx} = 2x$	$\frac{dy}{dx} = 1 + 2x$	$y := v_1 + x^2$
		$\frac{dy}{dx} = 2x$

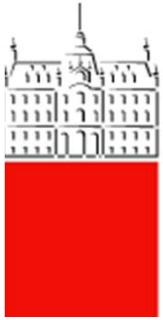
All the independent variables have to have an unique signature in order to prevent simplification A!

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["test", Real[x$$]];
x ⊢ SMSReal[x$$];
SMSD[Sin[x], x]
```

```
Cos [x]
```

```
<< AceGen`;
SMSInitialize["test"];
SMSModule["test", Real[x$$]];
x ⊢ SMSReal[x$$];
w ⊢ SMSFreeze[x^2 + 1];
SMSD[Sin[w], w]
```

```
Cos [w]
```

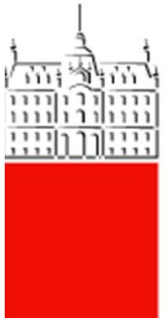


Automatic differentiation – Exceptions

AD produces values of derivatives of what is actually computed, rather than what one intends to compute.

- **How to translate mathematical formalisms into AD procedure?**
  - partial, total, directional, Lee, covariant, consistent, …
  - various notations $\frac{D(\bullet)}{D(\bullet)}$ $\frac{\partial(\bullet)}{\partial(\bullet)}$ $\delta(\bullet)$

- **Extended automatic differentiation technique (Korelc, 2002)**
  - Control of "exceptions" in AD is crucial to relate to mathematical formalisms



AD – Exception

Formalism for introduction of AD exceptions

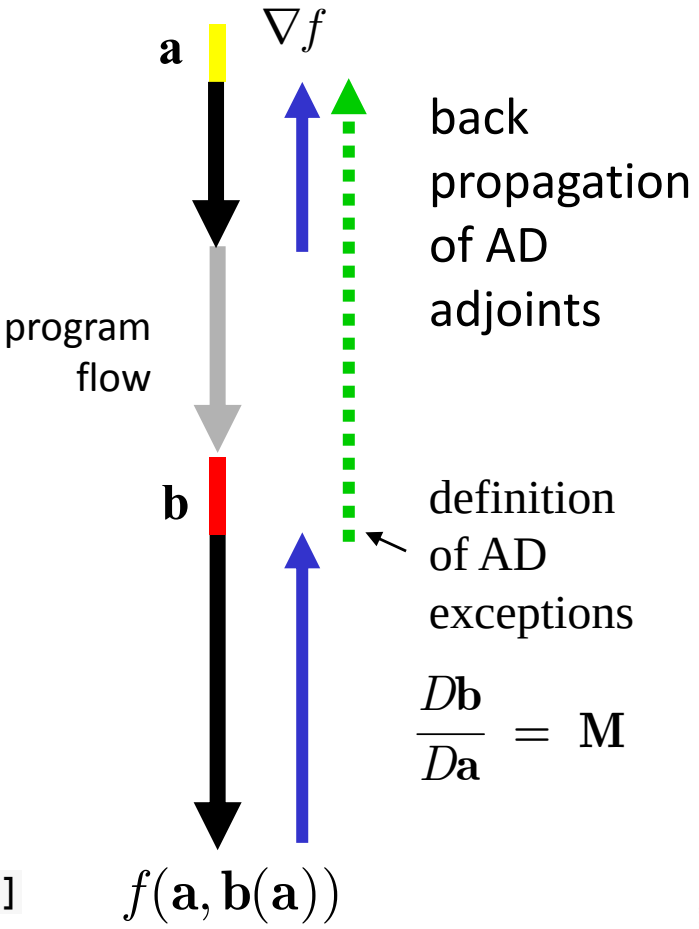
$\mathbf{a}$ ... independent variables

$\mathbf{b}$ ... intermediate variables

$f(\mathbf{a}, \mathbf{b}(\mathbf{a}))$ ... function

$\mathbf{M}$ ... arbitrary matrix

$$\nabla f := \left. \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}} \right|_{\frac{D\mathbf{b}}{D\mathbf{a}} = \mathbf{M}}$$



AceGen input - option "Dependency"

```
∇f ⊨ SMSD[f[a, b], a, "Dependency" → {b, a, M}]
```

Local/Global definition of AD exception

- Local AD exception
  - AD exception is introduced when AD procedure is executed

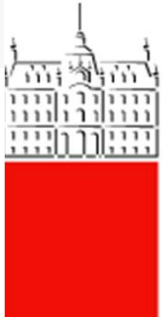
$$\nabla f_A := \left.\frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}}\right|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$$

```
a ⊢ SMSReal[a$$]
b ⊢ SMSFreeze[G[a]]
∇f ⊨ SMSD[f[a, b], a, "Dependency" → {b, a, M}]
```

- Global AD exception
  - AD exception is introducued together with intermediate variables $\mathbf{b}$

$$\mathbf{b} := \mathbf{G}(\mathbf{a})\Big|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$$
$$\nabla f_A := \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}}$$

```
a ⊢ SMSReal[a$$]
b ⊢ SMSFreeze[G[a], "Dependency" → {a, M}]
∇f ⊨ SMSD[f[a, b], a]
```



Types of AD exception

The basic situations that have to be considered are:

A. **Basic case:** The total derivatives of intermediate variables **a** with respect to independent variables **b** are set to be equal to matrix **M**.

$$\nabla f_A := \left.\frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta}\mathbf{a}}\right|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$$

B. **Special case:** There exists explicit dependency between variables that has to be neglected for the differentiation

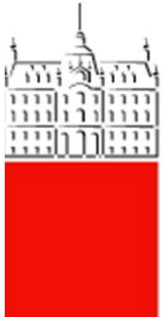
$$\nabla f_B := \left.\frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{a}))}{\hat{\delta}(\mathbf{a})}\right|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{0}}$$

C. **Implicit case:** There exists implicit dependency between variables that has to be considered for the differentiation

$$\nabla f_C := \left.\frac{\hat{\delta} f(\mathbf{a}, \mathbf{b})}{\hat{\delta}\mathbf{a}}\right|_{\frac{D\mathbf{b}}{D\mathbf{a}}=\mathbf{M}}$$

D. **Generalization:** The total derivatives of intermediate variables **a** with respect to intermediate variables **c** are set to be equal to matrix **M**.

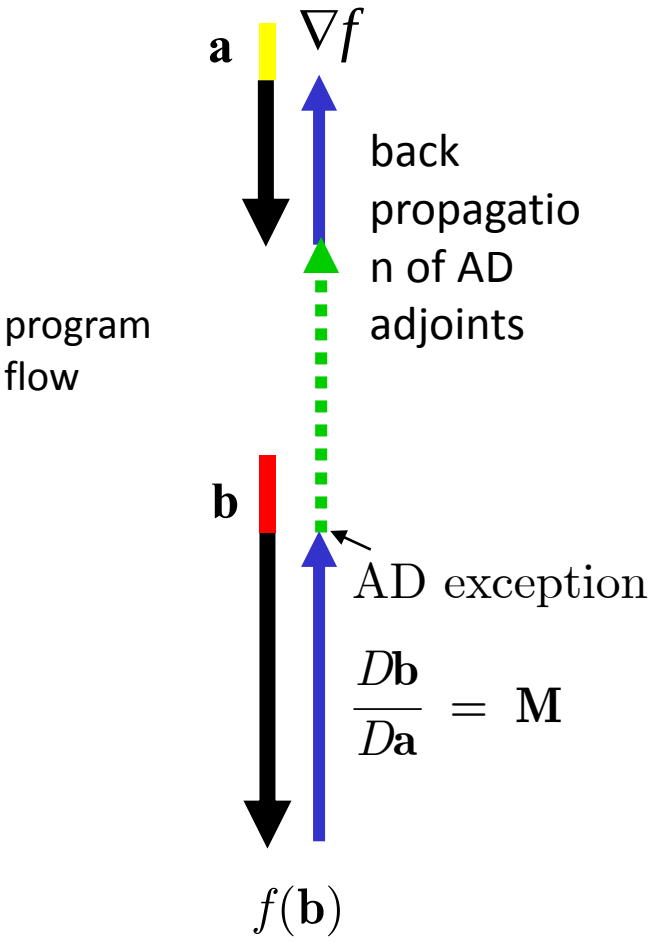
$$\nabla f_D := \left.\frac{\hat{\delta} f(\mathbf{a}, \mathbf{b}(\mathbf{c}(\mathbf{a})))}{\hat{\delta}\mathbf{a}}\right|_{\frac{D\mathbf{b}}{D\mathbf{c}}=\mathbf{M}}$$



AD – Exception Type C

- AD can "see" only explicit dependencies!
- implicit dependency between variables has to be specified as exception in AD

$$\nabla f := \left. \frac{\hat{\delta} f(\mathbf{a}, \mathbf{b})}{\hat{\delta}\mathbf{a}} \right|_{\frac{D\mathbf{b}}{D\mathbf{a}} = \mathbf{M}}$$



Example - exception Type C

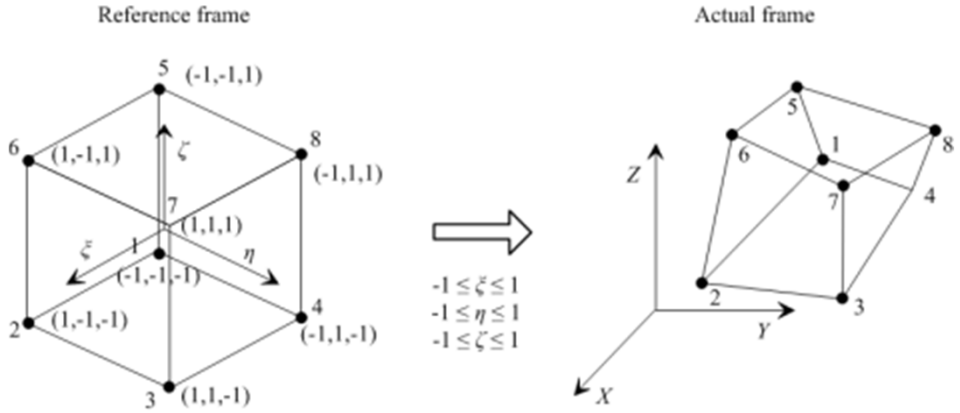
Nonlinear mapping from reference coordinates to initial coordinates in FEM

$\Xi = \{\xi, \eta, \zeta\}$ reference coordinates

$\mathbf{X}(\Xi) = \sum_k N(\Xi)_k \mathbf{X}_k$ actual coordinates

$\mathbf{u}(\Xi) = \sum_k N(\Xi)_k \mathbf{u}_k$ displacements

$\boldsymbol{H} = \dfrac{\partial \mathbf{u}}{\partial \mathbf{X}}$ displacement gradient



$$\boldsymbol{H} := \frac{\hat{\delta}\boldsymbol{u}(\Xi)}{\hat{\delta}\boldsymbol{X}(\Xi)} = 0 \qquad \text{wrong AD formulation}$$

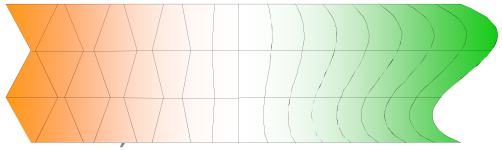
$$\boldsymbol{H} := \left.\frac{\hat{\delta}\boldsymbol{u}(\Xi)}{\hat{\delta}\boldsymbol{X}}\right|_{\frac{D\Xi}{D\mathbf{X}} = \left[\frac{\partial \mathbf{X}(\Xi)}{\partial \Xi}\right]^{-1}} \neq 0$$

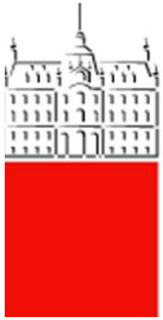
ADB notation
automatic differentiation based notation

Higher level symbolic language
(Mathematica + AceGen)

```
SMSD[u,X,"Dependency"->{Ξ,Inverse[SMSD[X,Ξ]}]
```

ADB form and AceFEM





Finite element solution procedure

1. strong form of boundary-value problem
2. weak form

$$\nabla(k\ \nabla\phi) + Q = 0$$

$$\delta\Pi = \int_\Omega \left(\nabla^T \delta\phi\ k\nabla\phi - \delta\phi\ Q\right) d\Omega - \ldots$$

$$\Pi = \int_\Omega \frac{1}{2}\left(\nabla\phi\right)^T\ k\nabla\phi d\Omega - \ldots$$

3. FE approximation of field variables
4. enforcement of local constraints
5. element quantities (**K**, **R**, …)
6. programming of steps 3, 4,5

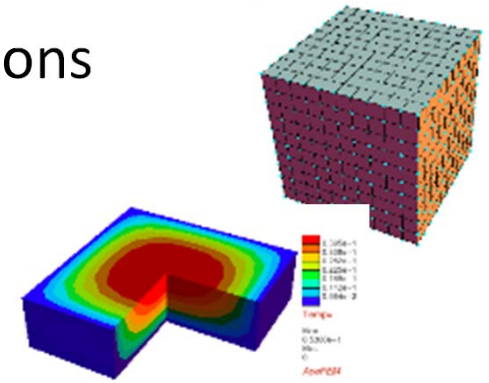
Automation of formulation at the individual element level

$$\phi = N_i \phi_i$$

$$R_i = \frac{\partial \Pi}{\partial \phi_i}$$

$$K_{i,j} = \frac{\partial R_i}{\partial \phi_j}$$

7. generation of mesh and boundary conditions
8. contact search algorithm
9. solution of the global problem
10. presentation and analysis of results

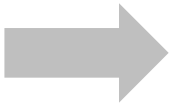


ADB Notation

The unification of the classical mathematical notation of computational models and the actual computer implementation can be achieved by means of automatic differentiation combined with the automatic code generation.

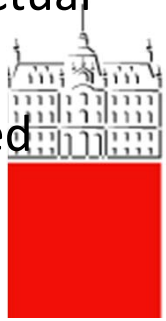
- ADB (Automatic Differentiation Based) form of computational model

Automatic differentiation
+
AD exceptions



ADB
(Automatic Differentiation Based)
form of computational model

- ADB form bridges mathematical notation of computational models and actual computer implementation.
- ADB form can be directly translated into the program code and the derived program code is numerically efficient.

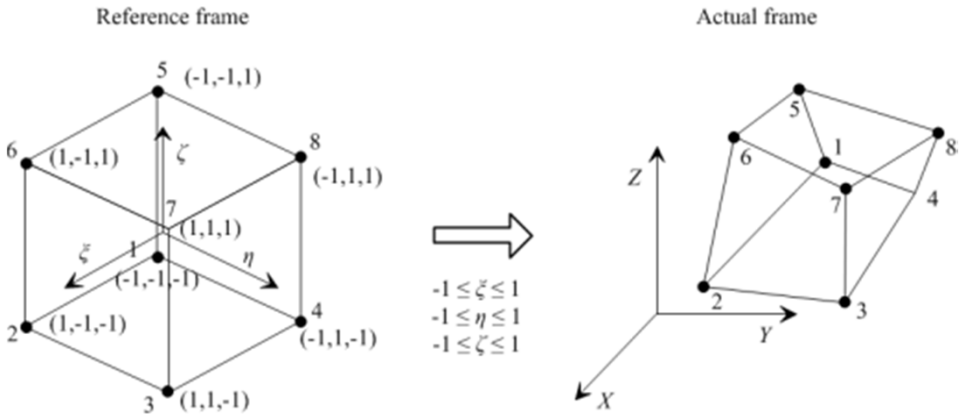


Example: 3D hyperelastic element

Problem is defined by:

- Hyperelastic strain energy function

$$W = \frac{\lambda}{2}(det\,\boldsymbol{F} - 1)^2 + \mu(\frac{(tr\,\boldsymbol{C} - 3)}{2} - Log(det\,\boldsymbol{F}))$$
$$\boldsymbol{F} = \mathbf{I} + \boldsymbol{H}$$
$$\boldsymbol{H} = \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{X}}$$



- FE approximation of coordinates and displacements
  - Nonlinear mapping from reference coordinates to initial coordinates

$\boldsymbol{\Xi} = \{\xi, \eta, \zeta\}$ reference coordinates

$\mathbf{X}(\boldsymbol{\Xi}) = \sum_k N(\boldsymbol{\Xi})_k \mathbf{X}_k$ actual coordinates

$\mathbf{u}(\boldsymbol{\Xi}) = \sum_k N(\boldsymbol{\Xi})_k \mathbf{u}_k$ displacements

Automation

$$\boldsymbol{H} := \frac{\hat{\delta}\boldsymbol{u}(\boldsymbol{\Xi})}{\hat{\delta}\boldsymbol{X}}\bigg|_{\frac{D\boldsymbol{\Xi}}{D\mathbf{X}} = \left[\frac{\partial \mathbf{X}(\boldsymbol{\Xi})}{\partial \boldsymbol{\Xi}}\right]^{-1}} \neq 0$$

$$\boldsymbol{H} := \frac{\hat{\delta}\boldsymbol{n}(\boldsymbol{\Xi})}{\hat{\delta}\boldsymbol{X}(\boldsymbol{\Xi})} = 0 \quad \text{wrong ADB notation}$$

Higher level symbolic language (Mathematica + AceGen)

```
SMSD[u,X,"Dependency"->{Ξ,Inverse[SMSD[X,Ξ]}]
```

ADB form: Hyperelastic material - A

$\mathbf{p}_e = \{p_1, p_2, ..., p_n\}$	vector of elements generalized d.o.f.
$\mathrm{W}(\mathbf{p}_e)$	hyperelastic strain energy function
$\mathbf{R}_e$	the contribution of the e-th element to the global residual
$\mathbf{R}_g$	integration point contribution of the residual of the e-th element

(A) Solution is a stationary point of the hyperelastic potential

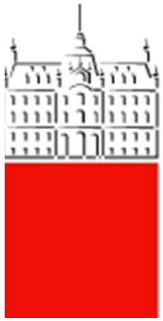
$$\Pi = \Pi^{\text{int}} + \Pi^{ext} \rightarrow \delta\Pi = 0$$

$$\Pi_e^{\text{int}} = \int_{\Omega_e} W\, dV \rightarrow \delta \int_{\Omega_e} W\, dV$$

$$\delta \int_{\Omega_e} W\, dV \approx \delta \mathbf{p}_e^T \left[\sum_g \left(J_g \frac{\partial W(\mathbf{p}_e)}{\partial \mathbf{p}_e} \right) w_g \right] = \delta \mathbf{p}_e^T \left[\sum_g \mathbf{R}_g w_g \right]$$

$$\mathbf{R}_e = \sum_g \mathbf{R}_g w_g$$

$$\mathbf{R}_g = J_g \frac{\partial W(\mathbf{p}_e)}{\partial \mathbf{p}_e} \quad \text{Automation} \quad \mathbf{R}_g := J_g \frac{\hat{\delta} W(\mathbf{p}_e)}{\hat{\delta} \mathbf{p}_e}$$



ADB form: Hyperelastic material - B

(B) Virtual work principle

$$\delta\Pi = \delta\Pi^{\text{int}} + \delta\Pi^{ext} = 0$$

$$\delta\Pi_e^{\text{int}} = \int_{\Omega_e} \mathbf{P}\cdot\delta\mathbf{F}\,dV \approx \sum_g \Big(J_g\,\mathbf{P}\cdot\delta\mathbf{F}\Big)w_g$$

$$\delta\mathbf{F} = \frac{\partial\mathbf{F}\left(\mathbf{p}_e\right)}{\partial\mathbf{p}_e}\delta\mathbf{p}_e$$

$$\delta\Pi_e^{\text{int}} \approx \delta\mathbf{p}_e^T\left[\sum_g J_g\mathbf{P}\cdot\frac{\partial\mathbf{F}\left(\mathbf{p}_e\right)}{\partial\mathbf{p}_e}w_g\right] = \delta\mathbf{p}_e^T\left[\sum_g \mathbf{R}_g w_g\right]$$

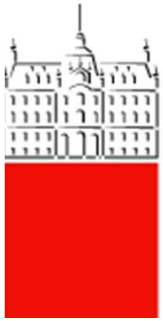
$$\mathbf{R}_g = J_g\,\mathbf{P}\cdot\frac{\partial\mathbf{F}\left(\mathbf{p}_e\right)}{\partial\mathbf{p}_e}$$

$$\mathbf{P} = \frac{\partial W}{\partial\mathbf{F}}$$

Automation

$$\mathbf{R}_g := J_g\,\mathbf{P}\cdot\frac{\hat{\delta}\mathbf{F}}{\hat{\delta}\mathbf{p}_e}$$

$$\mathbf{P} := \frac{\hat{\delta}W}{\hat{\delta}\mathbf{F}}$$



Numerical cost : Hyperelastic material

(A) Solution is a stationary point of the hyperelastic potential

$$\mathbf{R}_g := J_g \frac{\hat{\delta} W(\mathbf{p}_e)}{\hat{\delta} \mathbf{p}_e}$$

optimal ADB form

backward mode AD

$$\text{cost}(\mathbf{R}_g) \approx \alpha \, \text{cost}(W)$$

$$1.5 < \alpha < 5$$

(B) Virtual work principle

$$\mathbf{P} := \frac{\hat{\delta} W}{\hat{\delta} \mathbf{F}}$$

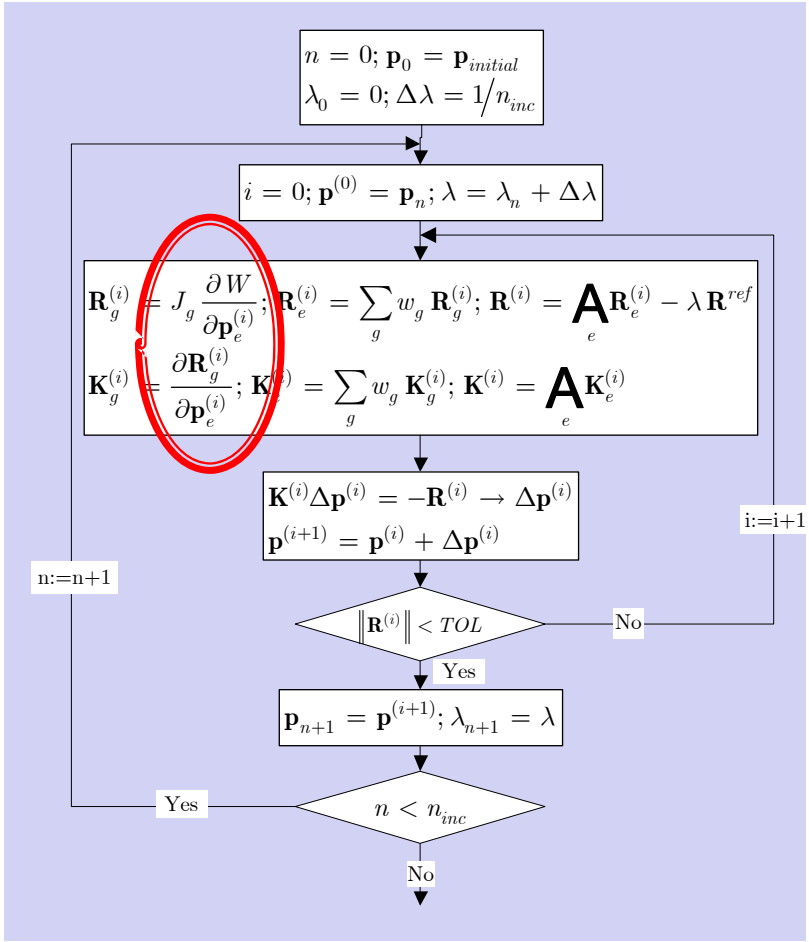
$$\mathbf{R}_g := J_g \, \mathbf{P} \cdot \frac{\hat{\delta} \mathbf{F}}{\hat{\delta} \mathbf{p}_e}$$

backward mode AD

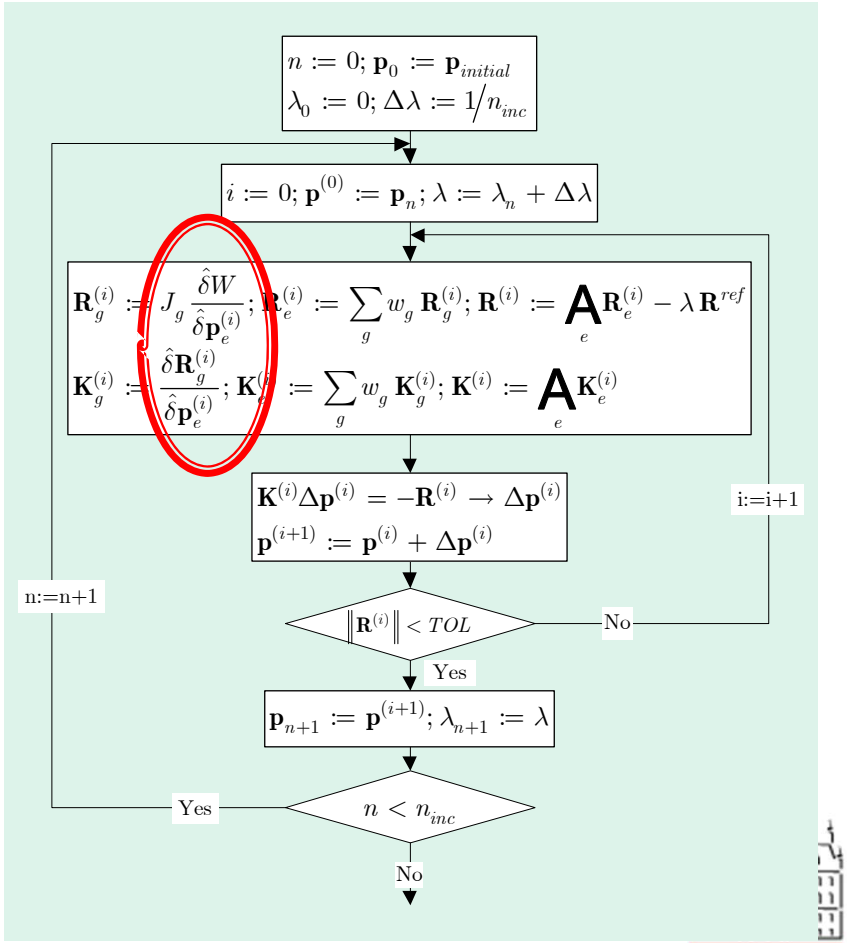
$$\text{cost}(\mathbf{R}_g) \approx \alpha \left(\text{cost}(W) + 9 \, \text{cost}(\mathbf{F}) \right)$$

Algorithm for primal analysis of hyperelastic problems

Newton-Raphson scheme for nonlinear hyperelastic problems
subjected to quasi-static proportional load
with constants load stepping



Automation of the scheme



Small strain plasticity - definitions

Model: elasto-plastic theory that assumes elastic isotropic response defined by the additive decomposition of strain tensor

- vector of element d.o.f. $\mathbf{p}_e$
- vector of unknown state variables $\mathbf{h}_g = \left\{\varepsilon^p, \lambda_m\right\}$ g - Gauss point
- elastic strain $\epsilon_e = \epsilon - \epsilon^p$
- elastic free energy function per unit volume $\mathrm{W}=\mathrm{W}(\varepsilon_e)$
- yield function $f(\sigma) = \sqrt{\frac{3}{2}\left(\sigma - \frac{1}{3}\mathrm{tr}(\sigma)\right) : \left(\sigma - \frac{1}{3}\mathrm{tr}(\sigma)\right)} - \sigma_y$
- evolution equations for ε^p

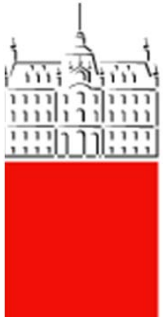
$$\epsilon^p - \epsilon_n^p - \Delta\varepsilon^p = 0$$

$$\Delta\varepsilon^p = \dot{\lambda}_m \frac{\partial f}{\partial \boldsymbol{\sigma}}$$

- $\mathbf{Q}$ - set of additional set of **algebraic equations** per Gauss point that has to be solved for unknown $\mathbf{h}_g$

$$\boldsymbol{Q} = \begin{Bmatrix} \epsilon^p - \epsilon_n^p - \Delta\varepsilon^p \\ f \end{Bmatrix} = 0$$

- Solution: local Newton-Raphson iterations at Gauss point
  - consequence: dependency of state variables on displacement $\mathbf{h}_g\left(\mathbf{p}_e\right)$



ADB form of small strain plasticity

Virtual work: $\mathbf{R}_g := \boldsymbol{\sigma} \cdot \dfrac{\hat{\delta}\boldsymbol{\varepsilon}}{\hat{\delta}\mathbf{p}_e}$

$\boldsymbol{\sigma}$

$f(\boldsymbol{\sigma}^{trial}) < 0$ elastic $\quad \boldsymbol{\sigma} := \dfrac{\hat{\delta}W(\boldsymbol{\varepsilon}, \mathbf{h}_{g\,n})}{\hat{\delta}\boldsymbol{\varepsilon}}$

$f(\boldsymbol{\sigma}^{trial}) \geq 0$ plastic $\quad \boldsymbol{\sigma} := \dfrac{\hat{\delta}W(\boldsymbol{\varepsilon}, \mathbf{h}_g, \mathbf{h}_{g,n})}{\hat{\delta}\boldsymbol{\varepsilon}}\Big|_{\frac{D\mathbf{h}_g}{D\mathbf{p}_e}=0}$

$$\frac{\hat{\delta}W}{\hat{\delta}\boldsymbol{\varepsilon}}\Big|_{\frac{D\mathbf{h}_g}{D\mathbf{p}_e}=0} \cdot \frac{\hat{\delta}\boldsymbol{\varepsilon}}{\hat{\delta}\mathbf{p}_e} = \frac{\hat{\delta}W}{\hat{\delta}\mathbf{p}_e}\Big|_{\frac{D\mathbf{h}_g}{D\mathbf{p}_e}=0}$$

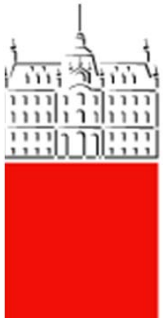
suppress implicit dependency due to the local NR loop

ADB form of plasticity problems

$$\mathbf{R}_g := J_g \frac{\hat{\delta}W}{\hat{\delta}\mathbf{p}_e}\Big|_{\frac{D\mathbf{h}_g}{D\mathbf{p}_e}=0}$$

$$\mathbf{K}_g := \frac{\hat{\delta}\mathbf{R}_g}{\hat{\delta}\mathbf{p}_e}\Big|_{\frac{D\mathbf{h}_g}{D\mathbf{p}_e}=-\left(\mathbf{A}_g\right)^{-1}\frac{\hat{\delta}\mathbf{Q}_g}{\hat{\delta}\mathbf{p}_e}}$$

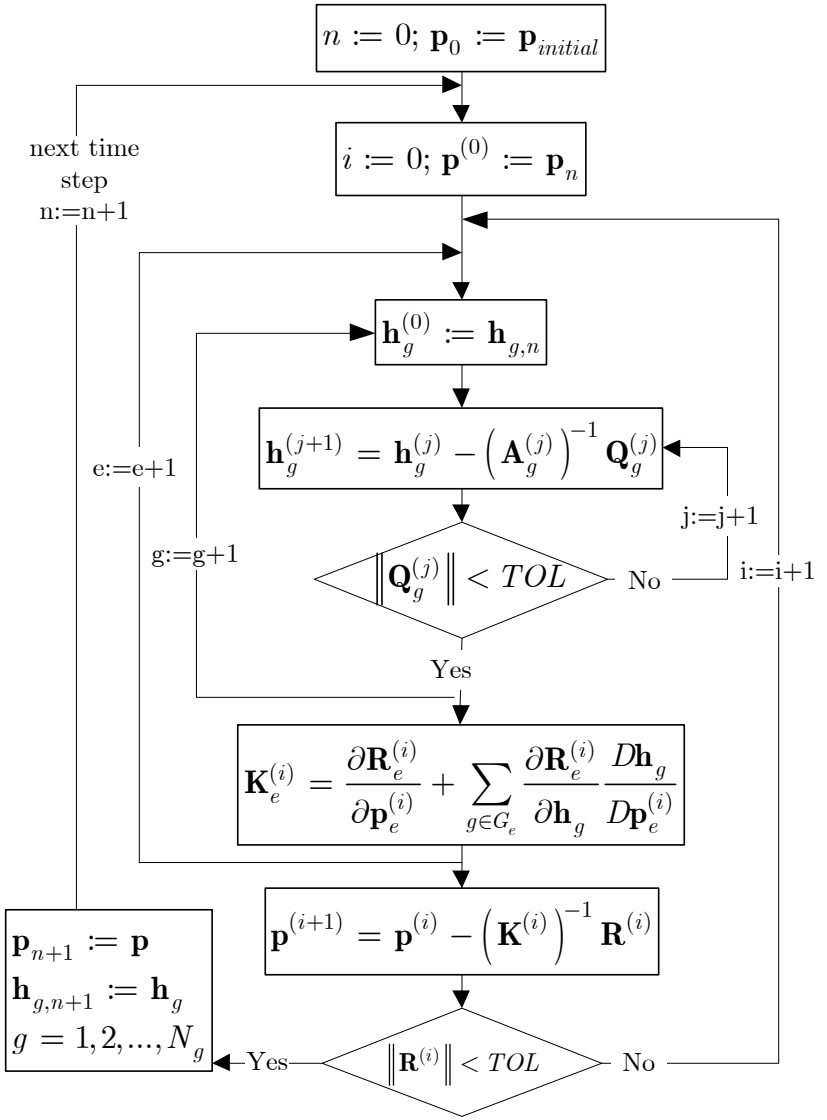
efficient form of consistent linearization (dependency due to the local NR loop)



KORELC, Jože. Automation of primal and sensitivity analysis of transient coupled problems. Comput. mech., 2009, 44:631-649.

Newton iterative procedure

Nested Newton iterative procedure



ADB form of tangent and residual

$\varepsilon := \varepsilon(\mathbf{p}_e)$

$f^{trial} := f(\varepsilon, \mathbf{h}_{g,n})$

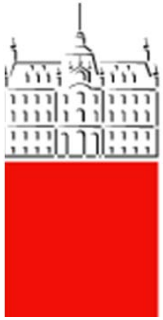
$f^{trial} \le 0 \left\{ \mathbf{h}_g := \mathbf{h}_{g,n} \right.$

$f^{trial} > 0 \left\{ \begin{array}{l} \mathbf{h}_g^{(0)} := \mathbf{h}_{g,n} \\ \text{repeat} \\ \quad \mathbf{A}_g^{(j)} := \dfrac{\delta \mathbf{Q}_g^{(j)}(\varepsilon, \mathbf{h}_g^{(j)}, \mathbf{h}_{g,n})}{\delta \mathbf{h}_g^{(j)}} \\ \quad \Delta \mathbf{h}_g^{(j)} := -\left(\mathbf{A}_g^{(j)}\right)^{-1} \mathbf{Q}_g^{(j)}(\varepsilon, \mathbf{h}_g^{(j)}, \mathbf{h}_{g,n}) \\ \quad \mathbf{h}_g^{(j+1)} := \mathbf{h}_g^{(j)} + \Delta \mathbf{h}_g^{(j)} \\ \text{until} \left\| \Delta \mathbf{h}_{g\,n+1}^{(j)} \right\| < TOL \\ \text{(*define global AD exception of type D for } \mathbf{h}_g \text{*)} \\ \mathbf{h}_g := \mathbf{h}_g \left|_{\frac{D\mathbf{h}_g}{D\varepsilon} = -\left(\mathbf{A}_g\right)^{-1} \frac{\delta \mathbf{Q}_g(\varepsilon, \mathbf{h}_g, \mathbf{h}_{g,n})}{\delta \varepsilon}} \right. \end{array}$

(*define local AD exception of type B for $\mathbf{h}_g$*)

$\mathbf{R}_g := J_g \dfrac{\delta W(\varepsilon, \mathbf{h}_g, \mathbf{h}_{g,n})}{\delta \mathbf{p}_e} \Big|_{\frac{D\mathbf{h}_g}{D\mathbf{p}_e} = \mathbf{0}}$

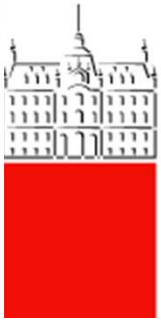
$\mathbf{K}_{Tg} := \dfrac{\delta \mathbf{R}_g}{\delta \mathbf{p}_e}$

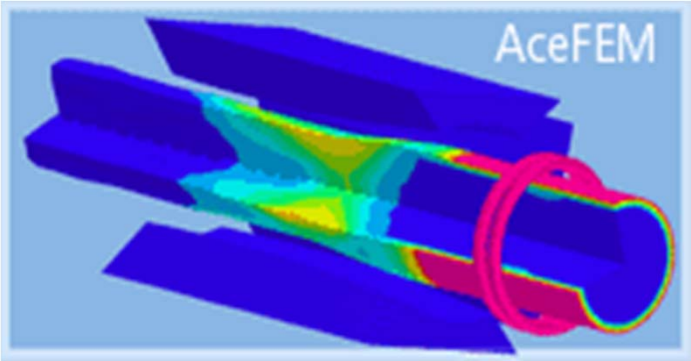


Comparison of code size and numerical efficiency

Element	Constitutive model	Code size (Kbytes)	Evaluation time (normalized)	*AceGen* time (normalized)	
Q1	linear elastic	9	1	1	
Q1	hyperelastic	9	1.6	1.3	
Q1	small strain elasto-plastic	24	3.0	7.4	
Q1	finite strain elasto-plastic	48	9.5	25	
Q1E4	linear elastic	10	1.6	2.11	2D
Q1E4	hyperelastic	15	3.4	3.5	
Q1E4	small strain elasto-plastic	27	3.7	12	
Q1E4	finite strain elasto-plastic	66	11.8	49	
H1	linear elastic	18	1	4.2	
H1	hyperelastic	21	1.5	4.5	
H1	small strain elasto-plastic	46	2.2	23.2	
H1	finite strain elasto-plastic	105	6.9	69.0	
H1E9	linear elastic	25	1.9	10.6	3D
H1E9	hyperelastic	46	4.3	16.5	
H1E9	small strain elasto-plastic	53	3.4	40.5	
H1E9	finite strain elasto-plastic	134	10.0	117.8	

- The presented comparison is based on an example where a rectangular bar is stretched, thus all the Gauss points are either in elastic or in plastic state.



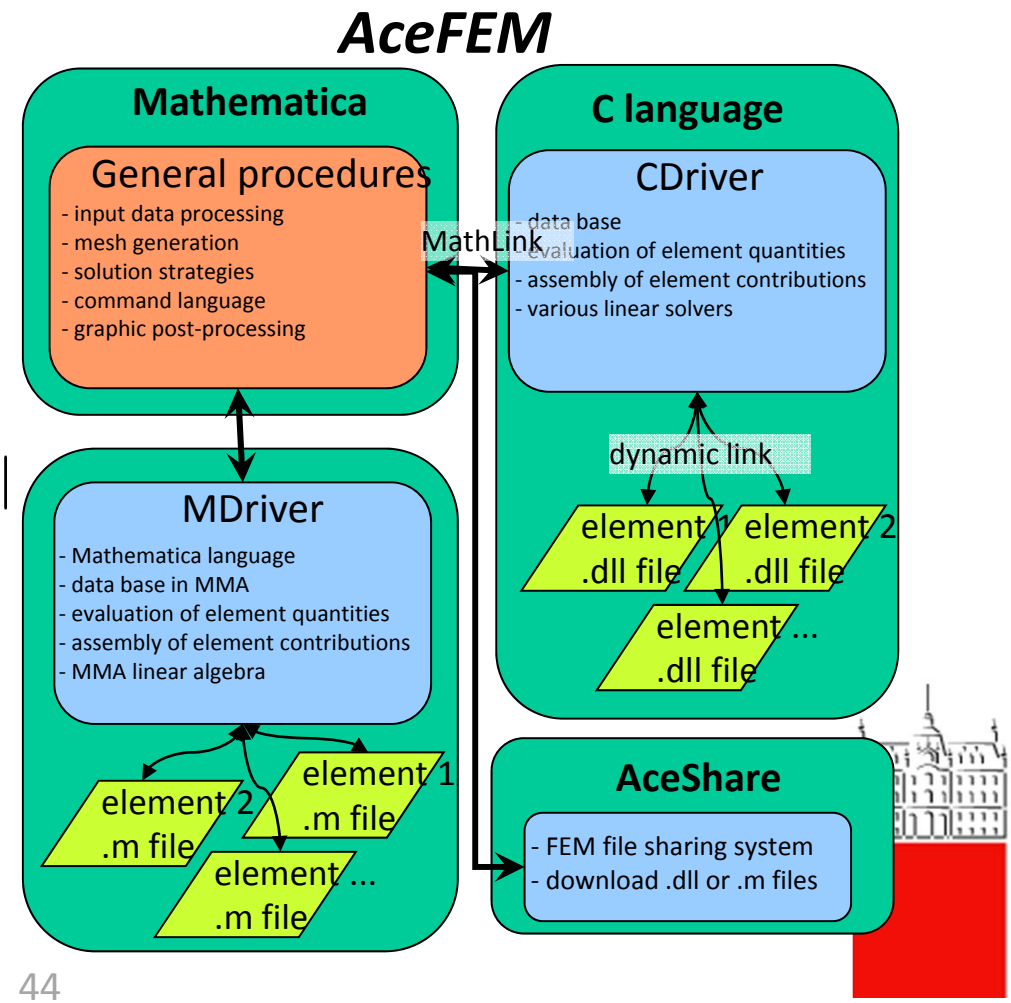


AceFEM

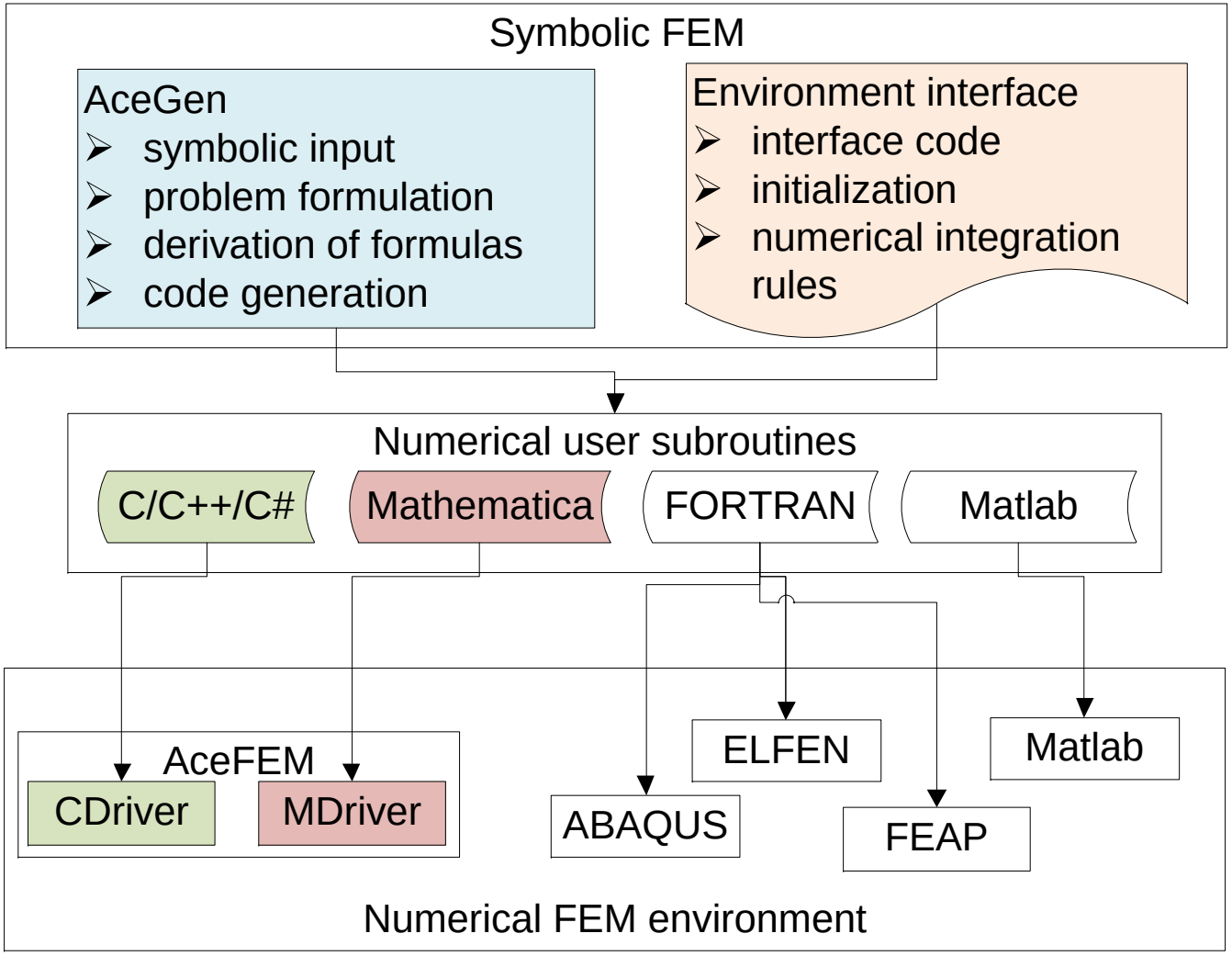
The Mathematica Finite Element Environment

Key features:

- hybrid symbolic-numeric FEM environment
- AceFEM combines use of Mathematica's features with external handling of intensive computations by compiled modules
- support for web-based FEM
- fast sparse solvers, exact sensitivity analysis, etc.



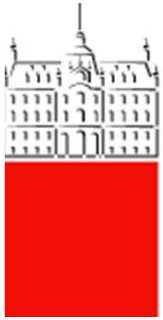
AceFEM and AceGEN



Data structures

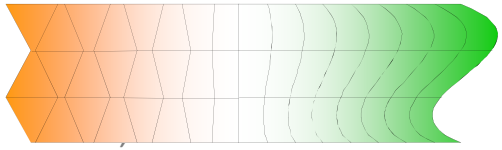
AceFEM data structures:

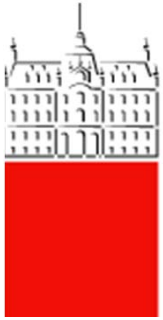
1. environment data defines a general information common to all nodes and elements **idata$$, rdata$$**
2. nodal data structure contains all the data that is associated with the node **nd$$**
3. node specification data structure contains information common for all nodes of particular type **ns$$**
4. element data structure contains all the data that is associated with the specific element **ed$$**
5. element specification data structure contains information common for all elements of particular type **es$$**



Advanced examples

- Debugging, verification, validation, ..
- Semi-analytical solution
- Optimization
- Coupled problems



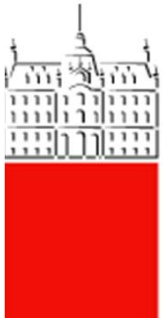


Verification & Validation of Numerical Codes

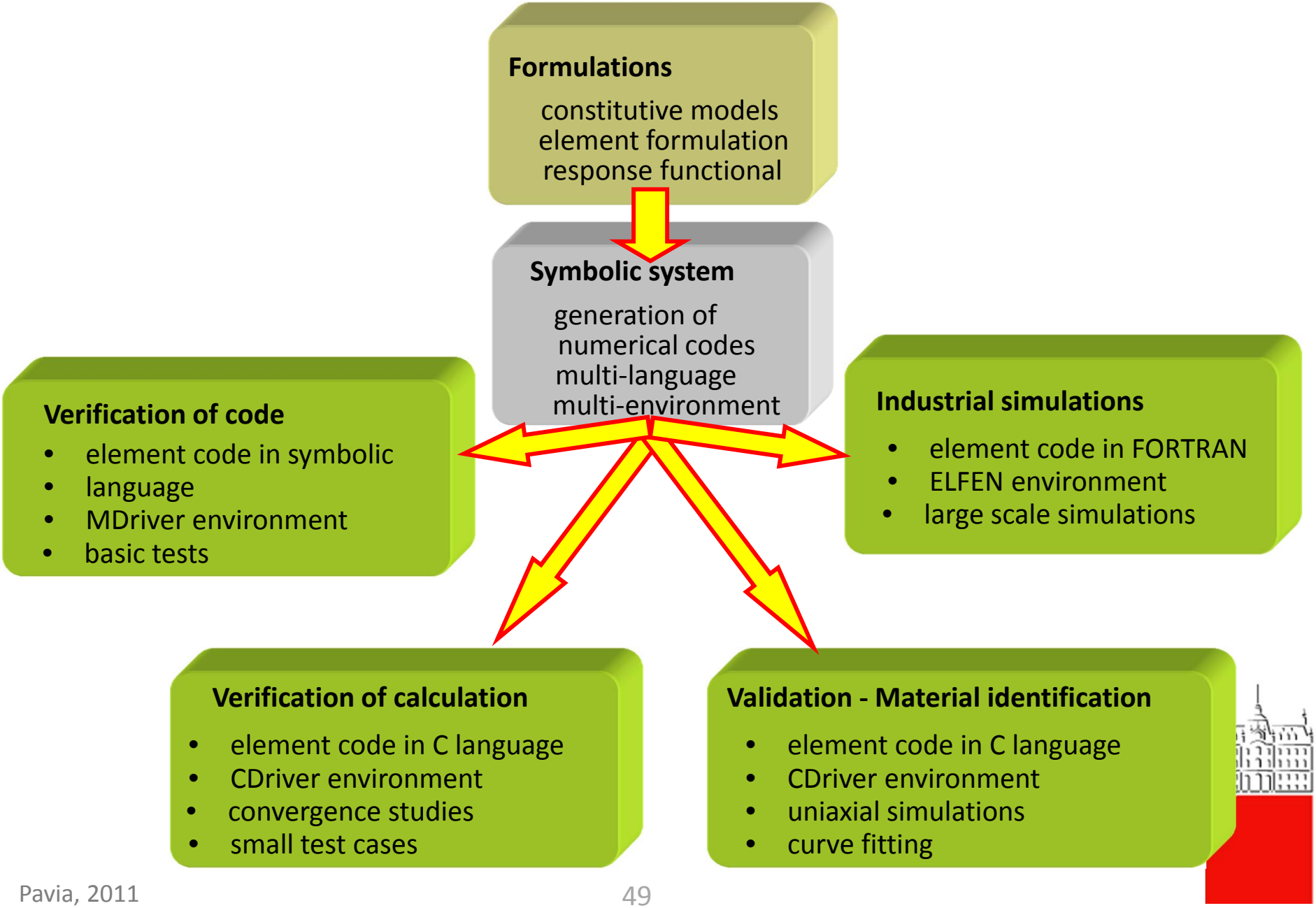
- Verification of Numerical Code
  - Is code correct?
  - benchmark tests (patch test, element eigenvalues tests, invariance tests)
  - code verification are ongoing activities of accumulating evidence that the code is correct

- Validation
  - Are the righ equations solved?
  - validation with more accurate physical models
  - validation with experiments

- Verification of Calculation
  - Are the equations solved correctly?
  - grid convergence studies relative to an unknown solution
  - error estimation



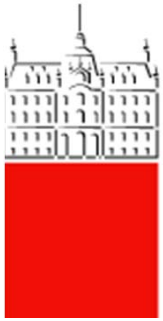
Advanced verifivation and validation procedure



Generic one element test

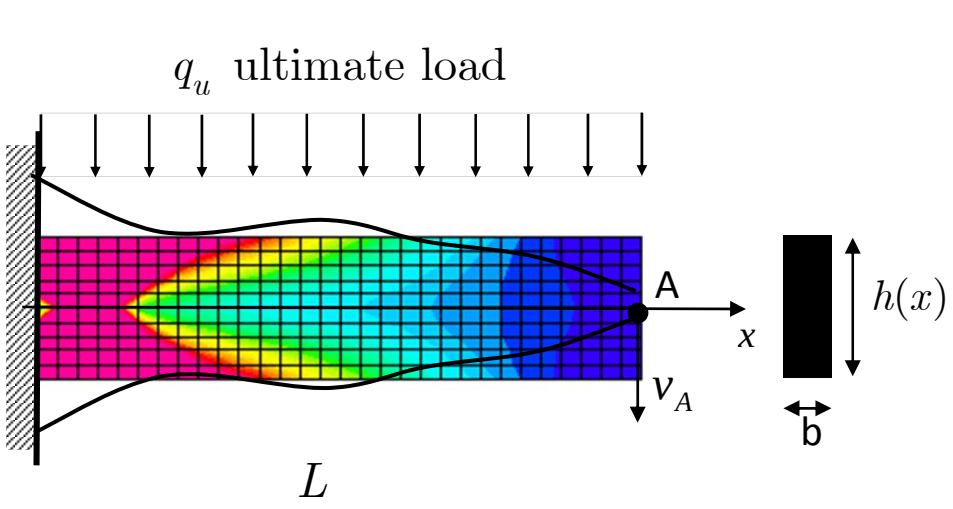
Generic one element test

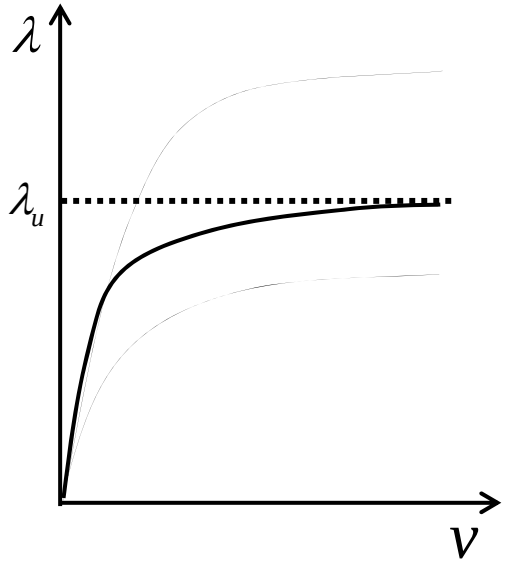
- to determine **convergence** characteristics of the element for the standard Newton-Raphson iterative scheme
- to apprise **behavior** of the element in constrained conditions as material **incompressibility** and extremely distorted or elongated element shapes
- to verify **objectivity** with respect to the superimposed rigid **body motion** on a deformed state of the element
- to verify **objectivity** with respect to the translation and rotation of the **reference** coordinate system
- analytical **sensitivity** analysis is independently verified by comparison with the finite difference method
- to verify the correctness of the automatically generated code when ported on various machines and for various finite element **environments**



Limit load optimisation of cantilever beam

- Task: find the shape of cantilever beam that has:
  - minimal volume
  - given ultimate load
  - ideal elasto-plastic material
  - 2D quadrilateral element





Formulation of the problem

- Three finite elemnts are needed to describe the problem:
  - 2D elasto plastic element
  - surface load element
  - prescribed displacement constrain element

$$\mathbf{p}_e = \{\boldsymbol{u}_e, \lambda\}$$

$\boldsymbol{u}_e$ displacements

λ load factor

$$\mathbf{R} = \begin{Bmatrix} \mathbf{R}_u \\ \mathbf{R}_\lambda \end{Bmatrix}$$

$$\begin{bmatrix} \mathbf{K}_{uu} & \mathbf{K}_{u\lambda} \\ \mathbf{K}_{\lambda u} & \mathbf{K}_{\lambda\lambda} \end{bmatrix} \cdot \begin{Bmatrix} \Delta \boldsymbol{u} \\ \Delta \lambda \end{Bmatrix} = \begin{Bmatrix} -\mathbf{R}_u \\ -\mathbf{R}_\lambda \end{Bmatrix}$$

$$\mathbf{R}_u = \mathbf{R}_u^{\text{el.-plast.}} + \mathbf{R}_u^{load}$$

$\mathbf{R}_u^{\text{el.-plast.}}$ elasto-plastic formulation equations

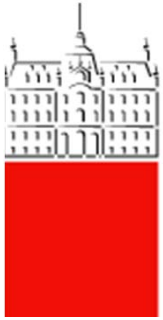
$$\mathbf{R}_{u_e}^{load} := -\left. \frac{\hat{\delta}\left(\int_\Omega \boldsymbol{T}(\lambda).\boldsymbol{u}\, d\Omega\right)}{\hat{\delta}\mathbf{p}_e} \right|_{\frac{D\boldsymbol{T}}{D\mathbf{p}_e}=\mathbf{0}}$$ load element equations

$$\mathbf{R}_\lambda := v_A - \gamma\, v_p$$ prescribed displacement constrain element equations

v_A displacement in node A

v_p prescdribed displacement in node A

γ path following parameter



Formulation of the problem

- Sensitivity problem - load element is path-independent

$$\mathbf{R}(\mathbf{p}(\phi),\phi) = \mathbf{0}$$

$$\mathbf{K}\frac{D\mathbf{p}}{D\phi} = \tilde{\mathbf{R}}$$

$$\tilde{\mathbf{R}} = -\frac{D\mathbf{R}}{D\phi} = \frac{D\mathbf{R}}{D\mathbf{X}}\frac{D\mathbf{X}}{D\phi} \quad \text{... shape sensitivity}$$

$$\frac{D\mathbf{X}}{D\phi} \quad \text{.... design velocity field - problem dependent}$$

$$\tilde{\mathbf{R}}_e := -\frac{\hat{\delta}(\mathbf{R}_e)}{\hat{\delta}\phi}\bigg|_{\frac{D\mathbf{X}}{D\phi}=DX} \quad \text{... element contribution}$$

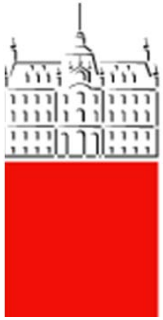
- Objective function

$$\min \Phi_0 \;\; ; \Phi_0 = w_1(\lambda - \lambda_u)^2 + w_2\mathbf{V}\text{olume} + w_3\sum_k \Phi_{\text{penalty constrain } h(x)>0}$$

λ_u prescribed limit load factor

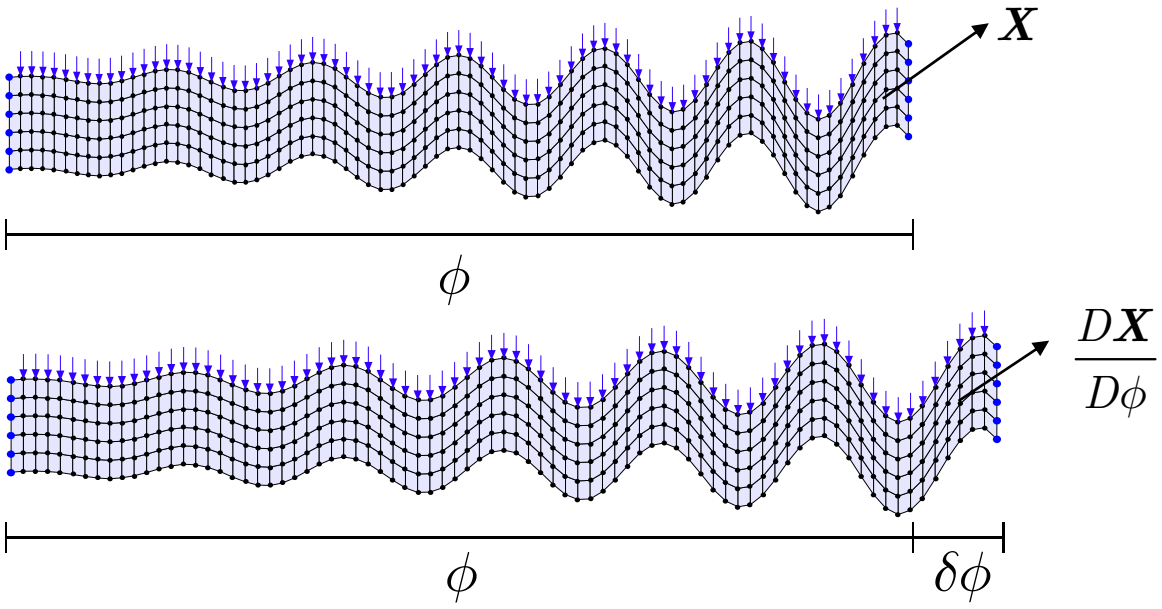
λ calculated limit load factor

w_1, w_2, w_3 weights



Design velocity field

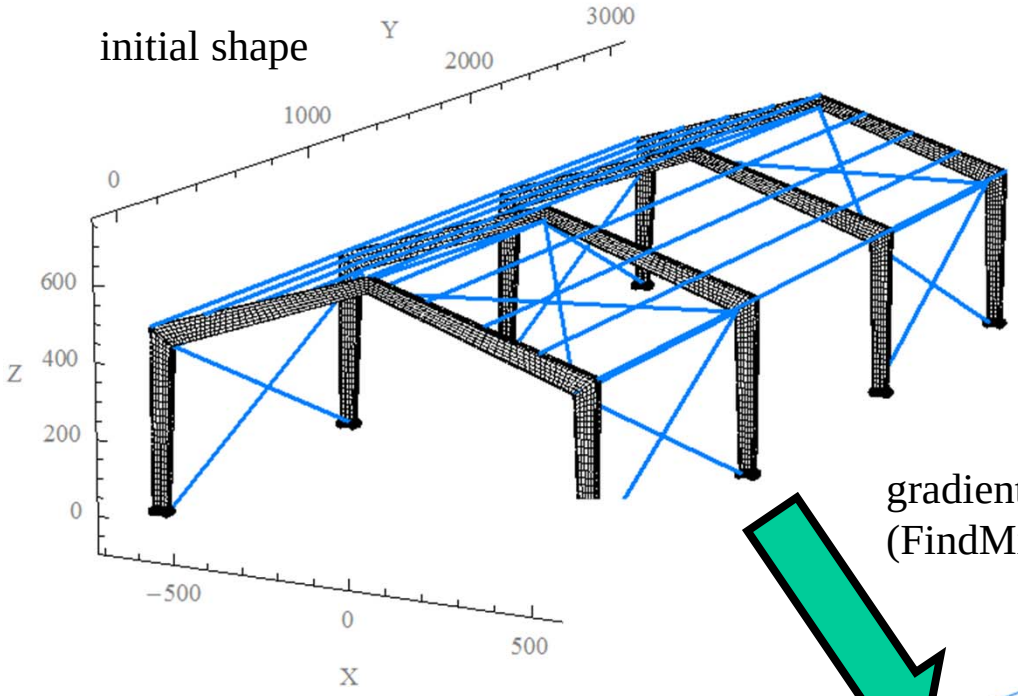
$\frac{D\boldsymbol{X}}{D\phi}$ .... design velocity field - problem dependent

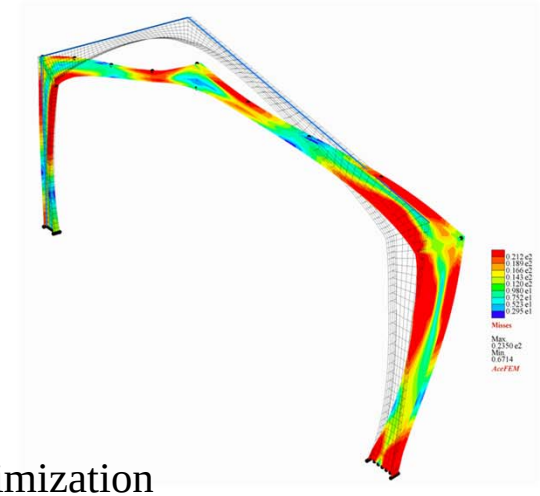


direct differentiation of symbolically parameterized mesh
based on hybrid symbolic-numeric AceFEM environment

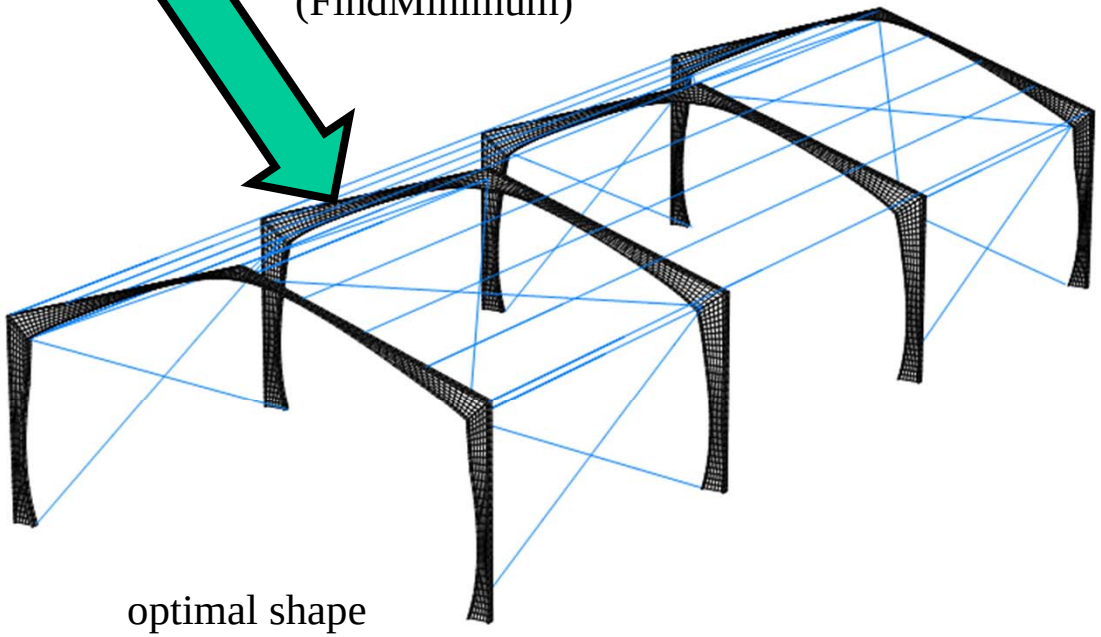
Large scale engineering optimisation

initial shape





gradient based optimization
(FindMinimum)



optimal shape