Automatic Structure Analysis

Stefan Klus and Slaven Peles

2014 TUG, Sandia National Laboratories

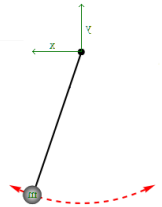
28 October 2014

United Technologies
Research Center

Solving nonlinear problems numerically

Typical solver requires model equations, Jacobian, and sparsity pattern to be provided by the modeler.

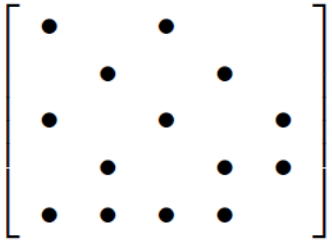
$$\begin{aligned} \dot{x} - v &= 0 \\ \dot{y} - w &= 0 \\ \dot{v} + \lambda x &= 0 \\ \dot{w} + \lambda y + 1 &= 0 \\ xv + yw &= 0 \end{aligned}$$



$$F(t, x, \dot{x}) = 0$$

$$W = \begin{bmatrix} \alpha & 0 & -\beta & 0 & 0 \\ 0 & \alpha & 0 & -\beta & 0 \\ \beta\lambda & 0 & \alpha & 0 & \beta x \\ 0 & \beta\lambda & 0 & \alpha & \beta y \\ \beta v & \beta w & \beta x & \beta y & 0 \end{bmatrix}$$

$$W = \alpha \frac{\partial F}{\partial \dot{x}} + \beta \frac{\partial F}{\partial x}$$



Model Equations

```
Epetra_Vector& f = *outArgs.get_f();
f[0] = -x_dot[0] + x[2];
f[1] = -x_dot[1] + x[3];
f[2] = -x_dot[2] - x[0]*x[4];
f[3] = -x_dot[3] - x[1]*x[4] - 1;
f[4] = x[0]*x[2] + x[1]*x[3];
```

Jacobian

```
int i0[] = {0, 2};
int i1[] = {1, 3};
int i2[] = {0, 2, 4};
int i3[] = {1, 3, 4};
int i4[] = {0, 1, 2, 3};
double d0[] = {-alpha, beta};
double d1[] = {-alpha, beta};
double d2[] = {-beta*x[4], -alpha, -beta*x[0]};
double d3[] = {-beta*x[4], -alpha, -beta*x[1]};
double d4[] = {beta*x[2], beta*x[3],
               beta*x[0], beta*x[1]};
crsW.ReplaceGlobalValues(0, 2, d0, i0);
crsW.ReplaceGlobalValues(1, 2, d1, i1);
crsW.ReplaceGlobalValues(2, 3, d2, i2);
crsW.ReplaceGlobalValues(3, 3, d3, i3);
crsW.ReplaceGlobalValues(4, 4, d4, i4);`
```

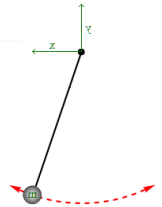
Sparsity pattern

```
int i0[] = {0, 2};
int i1[] = {1, 3};
int i2[] = {0, 2, 4};
int i3[] = {1, 3, 4};
int i4[] = {0, 1, 2, 3};
W_graph_->InsertGlobalIndices(0, 2, i0);
W_graph_->InsertGlobalIndices(1, 2, i1);
W_graph_->InsertGlobalIndices(2, 3, i2);
W_graph_->InsertGlobalIndices(3, 3, i3);
W_graph_->InsertGlobalIndices(4, 4, i4);
```

Solving nonlinear problems numerically

Typical solver requires model equations, Jacobian, and sparsity pattern to be provided by the modeler.

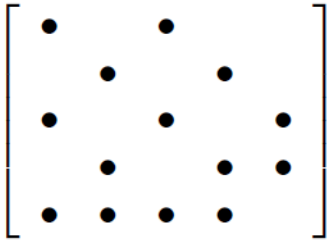
$$\begin{aligned} \dot{x} - v &= 0 \\ \dot{y} - w &= 0 \\ \dot{v} + \lambda x &= 0 \\ \dot{w} + \lambda y + 1 &= 0 \\ xv + yw &= 0 \end{aligned}$$



$$F(t, x, \dot{x}) = 0$$

$$W = \begin{bmatrix} \alpha & 0 & -\beta & 0 & 0 \\ 0 & \alpha & 0 & -\beta & 0 \\ \beta\lambda & 0 & \alpha & 0 & \beta x \\ 0 & \beta\lambda & 0 & \alpha & \beta y \\ \beta v & \beta w & \beta x & \beta y & 0 \end{bmatrix}$$

$$W = \alpha \frac{\partial F}{\partial \dot{x}} + \beta \frac{\partial F}{\partial x}$$



Model Equations

```
Epetra_Vector& f = *outArgs.get_f();
f[0] = -x_dot[0] + x[2];
f[1] = -x_dot[1] + x[3];
f[2] = -x_dot[2] - x[0]*x[4];
f[3] = -x_dot[3] - x[1]*x[4] - 1;
f[4] = x[0]*x[2] + x[1]*x[3];
```

Jacobian

```
int i0[] = {0, 2};
int i1[] = {1, 3};
int i2[] = {0, 2, 4};
int i3[] = {1, 3, 4};
int i4[] = {0, 1, 2, 3};
W_graph_->InsertGlobalIndices(0, 2, i0);
W_graph_->InsertGlobalIndices(1, 2, i1);
W_graph_->InsertGlobalIndices(2, 3, i2);
W_graph_->InsertGlobalIndices(3, 3, i3);
W_graph_->InsertGlobalIndices(4, 4, i4);
```

Automatic Differentiation (Sacado)

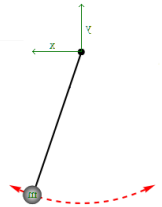
```
                beta*x[0], beta*x[1]};
crsW.ReplaceGlobalValues(0, 2, d0, i0);
crsW.ReplaceGlobalValues(1, 2, d1, i1);
crsW.ReplaceGlobalValues(2, 3, d2, i2);
crsW.ReplaceGlobalValues(3, 3, d3, i3);
crsW.ReplaceGlobalValues(4, 4, d4, i4);`
```

Sparsity pattern

Solving nonlinear problems numerically

Typical solver requires model equations, Jacobian, and sparsity pattern to be provided by the modeler.

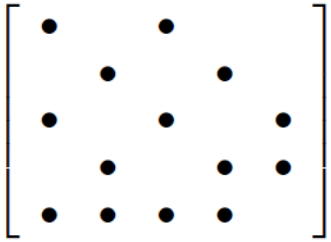
$$\begin{aligned} \dot{x} - v &= 0 \\ \dot{y} - w &= 0 \\ \dot{v} + \lambda x &= 0 \\ \dot{w} + \lambda y + 1 &= 0 \\ xv + yw &= 0 \end{aligned}$$



$$F(t, x, \dot{x}) = 0$$

$$W = \begin{bmatrix} \alpha & 0 & -\beta & 0 & 0 \\ 0 & \alpha & 0 & -\beta & 0 \\ \beta\lambda & 0 & \alpha & 0 & \beta x \\ 0 & \beta\lambda & 0 & \alpha & \beta y \\ \beta v & \beta w & \beta x & \beta y & 0 \end{bmatrix}$$

$$W = \alpha \frac{\partial F}{\partial \dot{x}} + \beta \frac{\partial F}{\partial x}$$



Model Equations

```
Epetra_Vector& f = *outArgs.get_f();
f[0] = -x_dot[0] + x[2];
f[1] = -x_dot[1] + x[3];
f[2] = -x_dot[2] - x[0]*x[4];
f[3] = -x_dot[3] - x[1]*x[4] - 1;
f[4] = x[0]*x[2] + x[1]*x[3];
```

Jacobian

```
int i0[] = {0, 2};
int i1[] = {1, 3};
                                                    eta*x[0]};
                                                    eta*x[1]};
                beta*x[0], beta*x[1]};
crsW.ReplaceGlobalValues(0, 2, d0, i0);
crsW.ReplaceGlobalValues(1, 2, d1, i1);
crsW.ReplaceGlobalValues(2, 3, d2, i2);
crsW.ReplaceGlobalValues(3, 3, d3, i3);
crsW.ReplaceGlobalValues(4, 4, d4, i4);`
```

Automatic Differentiation (Sacado)

Sparsity pattern

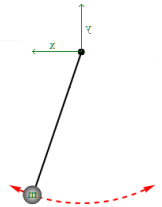
```
int i0[] = {0, 2};
int i1[] = {1, 3};
                                                    i0);
                                                    i1);
                                                    i2);
                                                    i3);
                                                    i4);
```

Automatic Structure Analysis

Solving nonlinear problems numerically

Typical solver requires model equations, Jacobian, and sparsity pattern to be provided by the modeler.

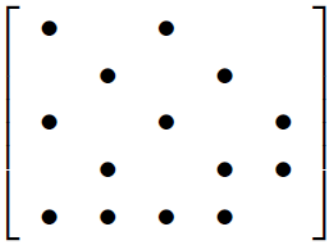
$$\begin{aligned} \dot{x} - v &= 0 \\ \dot{y} - w &= 0 \\ \dot{v} + \lambda x &= 0 \\ \dot{w} + \lambda y + 1 &= 0 \\ xv + yw &= 0 \end{aligned}$$



$$F(t, x, \dot{x}) = 0$$

$$W = \begin{bmatrix} \alpha & 0 & -\beta & 0 & 0 \\ 0 & \alpha & 0 & -\beta & 0 \\ \beta\lambda & 0 & \alpha & 0 & \beta x \\ 0 & \beta\lambda & 0 & \alpha & \beta y \\ \beta v & \beta w & \beta x & \beta y & 0 \end{bmatrix}$$

$$W = \alpha \frac{\partial F}{\partial \dot{x}} + \beta \frac{\partial F}{\partial x}$$



Model Equations

```
Epetra_Vector& f = *outArgs.get_f();
f[0] = -x_dot[0] + x[2];
f[1] = -x_dot[1] + x[3];
f[2] = -x_dot[2] - x[0]*x[4];
f[3] = -x_dot[3] - x[1]*x[4] - 1;
f[4] = x[0]*x[2] + x[1]*x[3];
```

Jacobian

```
int i0[] = {0, 2};
int i1[] = {1, 3};
                                    eta*x[0]};
                                    eta*x[1]};
                [0], beta*x[1]};
crsW.ReplaceGloba  ues(0, 2, d0, i0);
crsW.ReplaceGloba  ues(1, 2, d1, i1);
crsW.ReplaceGlobalValues(2, 3, d2, i2);
```

Sparsity pattern

```
int i0[] = {0, 2};
int i1[] = {1, 3};
                                    i0);
                                    i1);
                                    i2);
                                    i3);
                                    i4);
```

Automatic Differentiation (Sacado)

Automatic Structure Analysis

Sparse Automatic Differentiation

Computing Jacobians

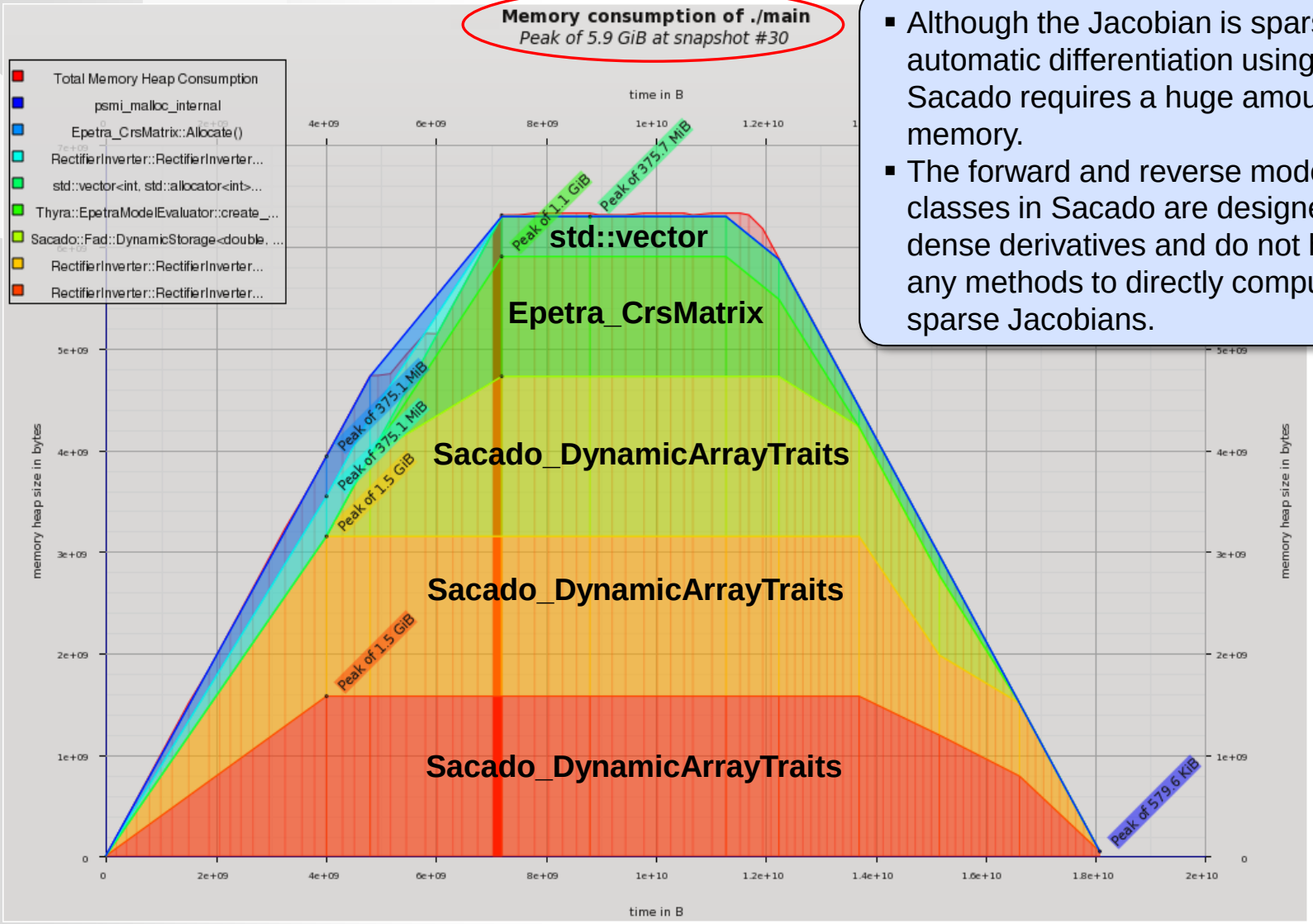
Evaluating derivatives of model equations accurately is key for successful simulations.

There are different ways to compute Jacobians:

- Numerical approximation:
  - First-order approximation: $\frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x)}{h_i}$
  - Requires *n* additional function evaluations.
  - Sensitive to choice of h_i.
  - Higher order approximations are more accurate, but also more costly.
- Analytic Jacobian:
  - Implement all derivatives manually.
  - Exact results.
  - Time-consuming and error-prone.
- Automatic differentiation:
  - The Jacobian is computed automatically every time the function f is evaluated.
  - Exact results.
  - Uses different data types (AD double objects), functions have to be rewritten to support automatic differentiation.
  - Computational cost: approximately 5 function evaluations (reverse mode AD).
  - Mature and well designed solutions such as **Sacado** already exist.

Limitations of Sacado

Example: Memory use of simulation of DC bus with 1000 AC loads.



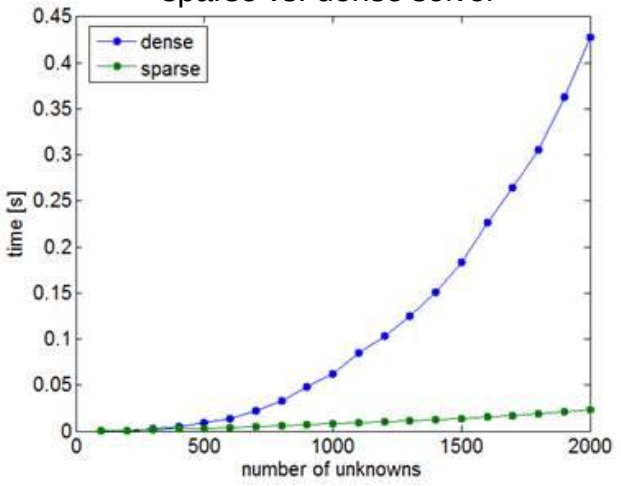
- Although the Jacobian is sparse, automatic differentiation using Sacado requires a huge amount of memory.
- The forward and reverse mode AD classes in Sacado are designed for dense derivatives and do not have any methods to directly compute sparse Jacobians.

Automatic sparse structure analysis

Analyze the structure of model equations and automatically create dependency graphs.

- Used to generate sparse Jacobian required for efficient simulations.
- Enables advanced analysis techniques such as index reduction, causalization, diagnostics, etc.
- Additional requirements:
  - Parameter/variable designation not known at compile time.
  - Sparsity pattern not known at compile time.
  - Model can be modified at runtime by the user.

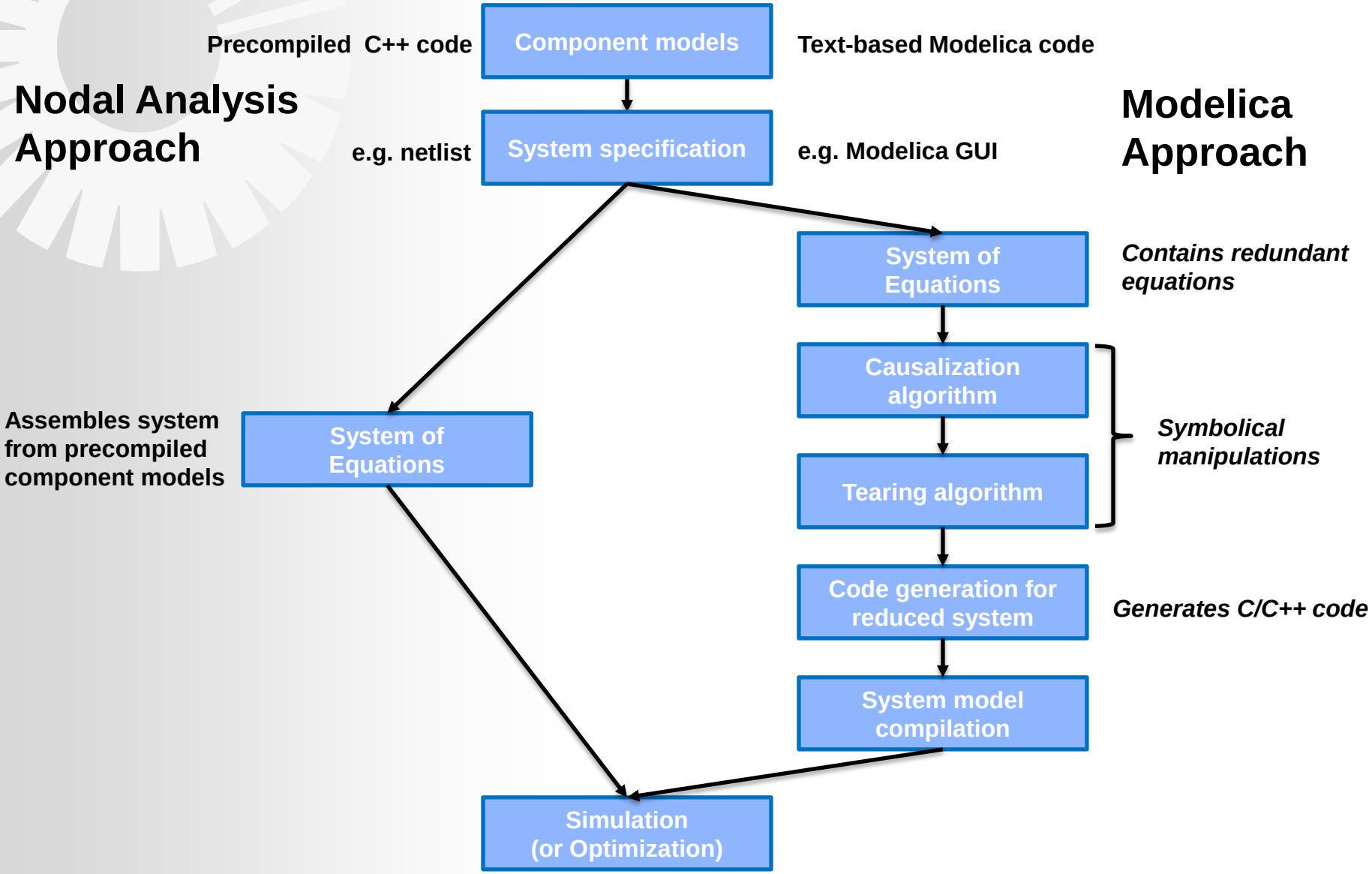
Computational cost scaling:
sparse vs. dense solver



Current Solutions

- **Modelica approach**: Recompile hard-wired model on the fly whenever user enters changes.
- **Extended nodal analysis approach**: Create dynamic model that is not hard-wired and that user can easily modify at runtime.

Comparison with Modelica-based approach



Automatic structure analysis implementation

Operator overloading approach similar to Sacado implementation.

Class definition and operators

```
class Variable
{
public:
    Variable();
    explicit Variable(double value);
    Variable(double value, size_t variableNumber);
    Variable(const Variable& v);
    ~Variable();

    // =
    Variable& operator=(const double& rhs);
    Variable& operator=(const Variable& rhs);

    // +=
    Variable& operator+=(const double& rhs);
    Variable& operator+=(const Variable& rhs);

    // -=
    Variable& operator-=(const double& rhs);
    Variable& operator-=(const Variable& rhs);

    // *=
    Variable& operator*=(const double& rhs);
    Variable& operator*=(const Variable& rhs);

    // /=
    Variable& operator/=(const double& rhs);
    Variable& operator/=(const Variable& rhs);

    // ...

private:
    double value_;
    size_t variableNumber_;
    bool isFixed_;

    mutable DependencyMap* dependencies_;
};
```

Mathematical functions

```
// math functions
#define DEFINE_FUN_1(fun)             \
    Variable fun(const Variable& x);
DEFINE_FUN_1(sin)
DEFINE_FUN_1(cos)
DEFINE_FUN_1(tan)
DEFINE_FUN_1(acos)
DEFINE_FUN_1(asin)
DEFINE_FUN_1(atan)
DEFINE_FUN_1(cosh)
DEFINE_FUN_1(sinh)
DEFINE_FUN_1(tanh)
DEFINE_FUN_1(exp)
DEFINE_FUN_1(log)
DEFINE_FUN_1(log10)
DEFINE_FUN_1(sqrt)
DEFINE_FUN_1(abs)
#undef DEFINE_FUN_1

#define DEFINE_FUN_2(fun)                                   \
    Variable fun(const Variable& x, const Variable& y);    \
    Variable fun(const Variable& x, double y);             \
    Variable fun(double x,          const Variable& y);

DEFINE_FUN_2(pow)
DEFINE_FUN_2(atan2)
DEFINE_FUN_2(min)
DEFINE_FUN_2(max)
#undef DEFINE_FUN_2
```

Example

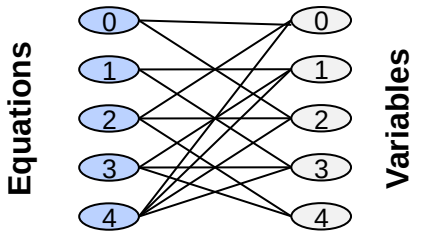
```
void residualFunction(VariableVector& f,
                      VariableVector& x,
                      VariableVector& x_dot)
{
    f[0] = -x_dot[0] + x[2];
    f[1] = -x_dot[1] + x[3];
    f[2] = -x_dot[2] - x[0]*x[4];
    f[3] = -x_dot[3] - x[1]*x[4] - 1;
    f[4] = x[0]*x[2] + x[1]*x[3];
}

int main()
{
    const size_t n = 5;

    StdVector<Variable> x(n), x_dot(n), f(n);
    for (size_t i = 0; i < n; ++i)
    {
        x[i].setVariableNumber(i);
        x_dot[i].setVariableNumber(i);
    }
    residualFunction(f, x, x_dot);
    printIncidenceMatrix(f);
}
```

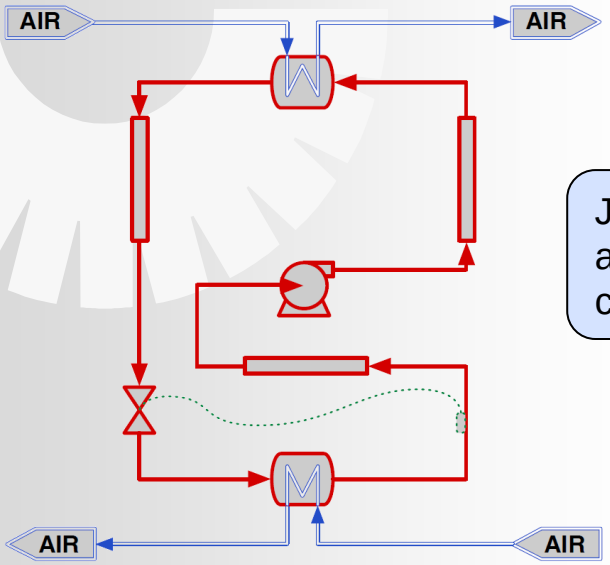
Result

```
A = [1 0 1 0 0;
     0 1 0 1 0;
     1 0 1 0 1;
     0 1 0 1 1;
     1 1 1 1 0];
```

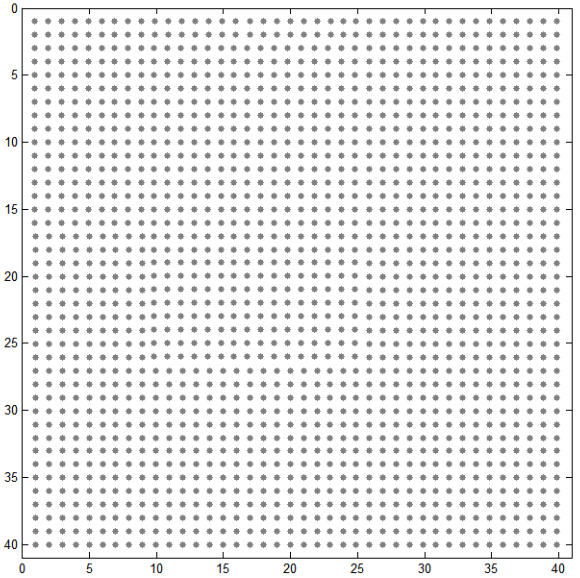


Automatic Jacobian computation

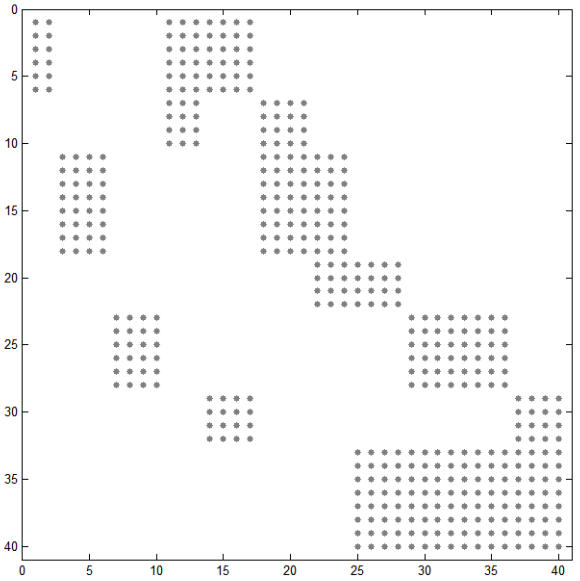
Dense vs. block sparse vs. sparse.



Jacobian structure for a simple vapor compression model.



Dense Jacobian

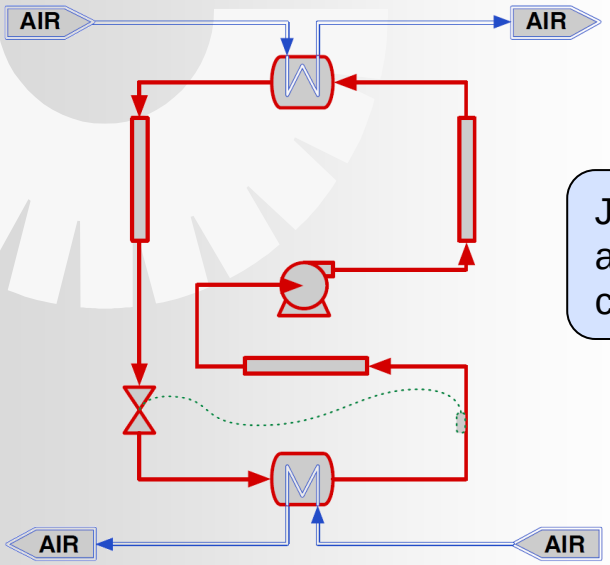


Sparse Jacobian

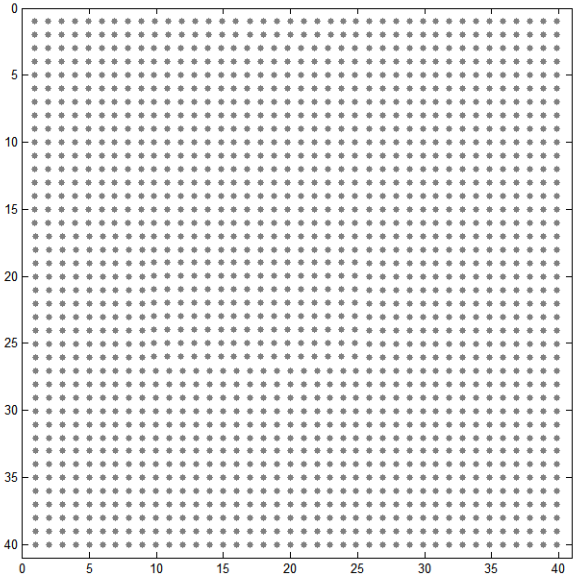
- Sacado could be used to compute dense Jacobians on a component-by-component basis. This results in a “block sparse” Jacobians.
- For large components, the overhead can be significant and will slow down the solver.
- Sparse Jacobians maximize utilization of state-of-the-art solvers in NOX and improve simulation speed.

Automatic Jacobian computation

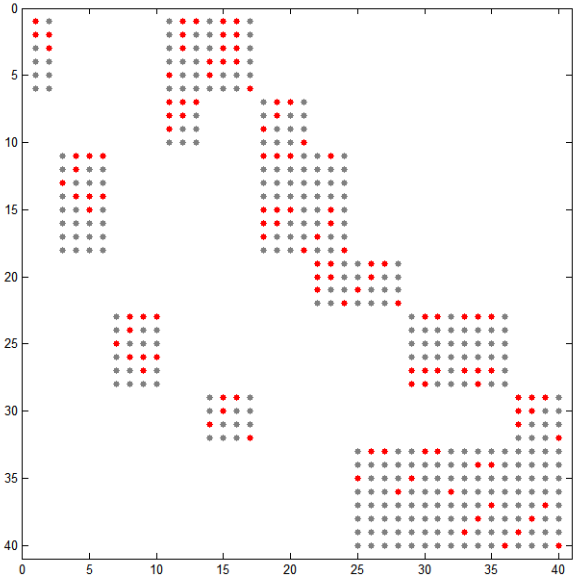
Dense vs. block sparse vs. sparse.



Jacobian structure for a simple vapor compression model.



Dense Jacobian

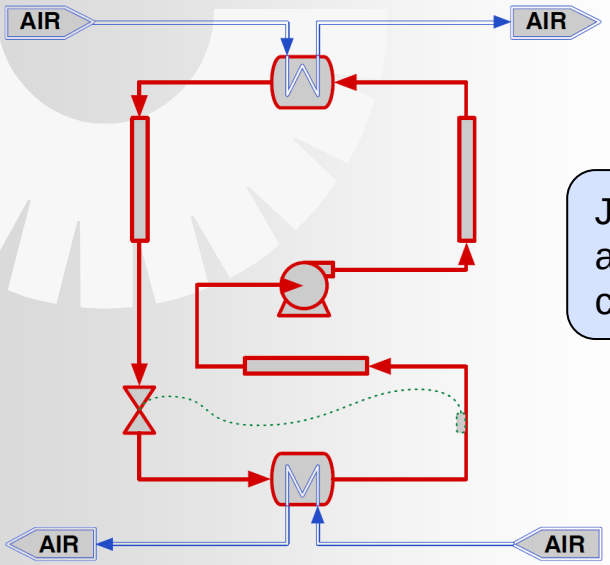


Sparse Jacobian

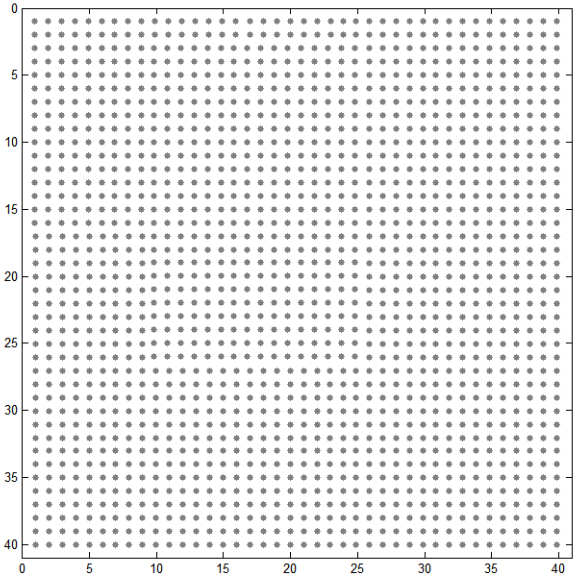
- Sacado could be used to compute dense Jacobians on a component-by-component basis. This results in a “block sparse” Jacobians.
- For large components, the overhead can be significant and will slow down the solver.
- Sparse Jacobians maximize utilization of state-of-the-art solvers in NOX and improve simulation speed.

Automatic Jacobian computation

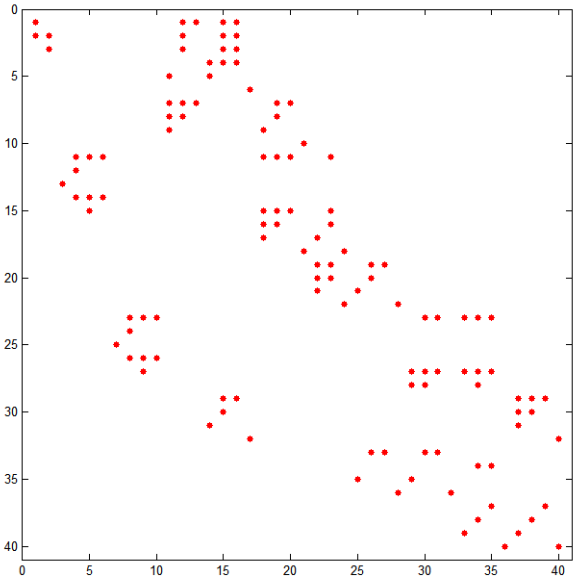
Dense vs. block sparse vs. sparse.



Jacobian structure for a simple vapor compression model.



Dense Jacobian

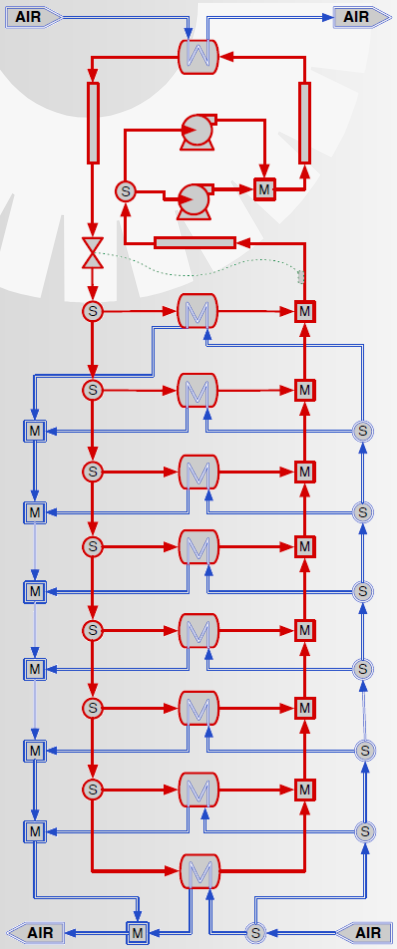


Sparse Jacobian

- Sacado could be used to compute dense Jacobians on a component-by-component basis. This results in a “block sparse” Jacobians.
- For large components, the overhead can be significant and will slow down the solver.
- Sparse Jacobians maximize utilization of state-of-the-art solvers in NOX and improve simulation speed.

Capability demonstration

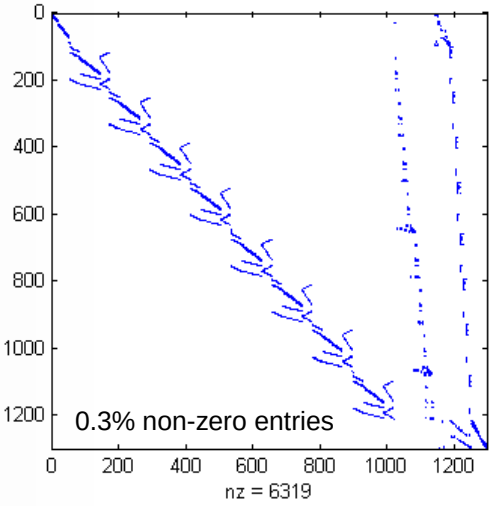
Automatic Jacobian computation and simulation of a vapor compression system.



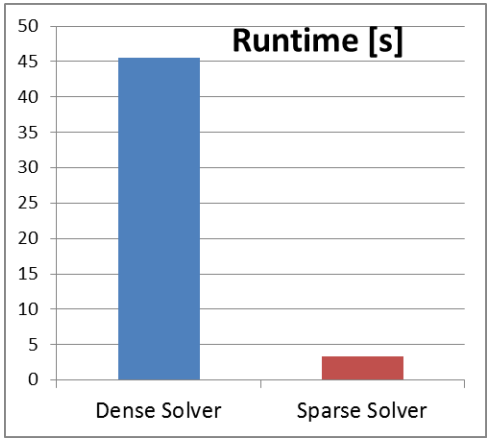
Problem Size
1298 Residual equations
15 Finite elements in each evaporator model

- Accuracy of Jacobian computation verified against dense numerical Jacobian (1st order finite difference).
- Preliminary scaling results show smaller overhead than for a block sparse Jacobian computation.
- All model configurations such as parameter and variable designation done at *runtime*.
- Redundant vapor compression system used as a test case.

Sparsity pattern
(non-zero elements)



Performance comparison with dense method



Open questions and next steps

- For the computation of sparse Jacobians using automatic differentiation, the two data types *Variable* and Sacado's *DoubleAD* should be combined.
- The *Variable* class does not require any other files or libraries (except for `boost::foreach`, which can be replaced) and could be easily integrated.
- Code not optimized. More profiling required to further improve performance scaling.
- Physics based models need to support automatic differentiation. Some practices used in legacy codes need to be revised for this approach to work.
  - How to incorporate look-up tables?
  - How can conditional statements be handled if different branches depend on different variables?
    - Currently, dependencies for different branches are added manually.
    - This process should be automated, i.e. equations should be written in such a way that dependencies are tracked automatically.
  - How to handle (eliminate?) embedded solvers in component models?
- Epetra does not support *DoubleAD* vectors or matrices, that means that entries need to be copied. This introduces additional overhead. Tpetra?