

SAND2006–2256
Unlimited Release
Printed April 2006

Nonlinear algebraic multigrid for constrained solid mechanics problems using *Trilinos*

Michael W. Gee

Computational Math & Algorithms
Sandia National Laboratories
PO Box 5800, Albuquerque, NM, 87185–1110

Ray S. Tuminaro

Computational Math & Algorithms
Sandia National Laboratories
PO Box 0969, Livermore, CA, 94551–0969

Key words nonlinear multigrid, algebraic multigrid, smoothed aggregation, nonlinear systems of equations, full approximation scheme, nonlinear conjugate gradients, Newton’s method

Abstract

The application of the finite element method to nonlinear solid mechanics problems results in the necessity to repeatedly solve a large nonlinear set of equations. In this paper we limit ourself to problems arising in constrained solid mechanics problems. It is common to apply some variant of Newton’s method or a Newton–Krylov method to such problems. Often, an analytic Jacobian matrix is formed and used in the above mentioned methods. However, if no analytic Jacobian is given, Newton methods might not be the method of choice. Here, we focus on a variational nonlinear multigrid approach that adopts the smoothed aggregation algebraic multigrid method to generate a hierarchy of coarse grids in a purely algebraic manner. We use preconditioned nonlinear conjugate gradient methods and/or quasi–Newton methods as nonlinear smoothers on fine and coarse grids. In addition we discuss the possibility to augment this basic algorithm with an automatically generated Jacobian by applying a block colored finite differencing scheme. After outlining the fundamental algorithms we give some examples and provide documentation for the parallel implementation of the described method within the *Trilinos* framework.

Acknowledgment

This work was partially funded by the Department of Energy Office of Science MICS program at Sandia National Laboratory. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE–AC04–94AL85000.

(page intentionally left blank)

1 Introduction and problem definition

The application of the finite element method to nonlinear solid mechanics problems results in the necessity to repeatedly solve a nonlinear function

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}, \quad (1)$$

where $\mathbf{F} : \mathbb{R}^N \rightarrow \mathbb{R}^N$ usually is a vector of residual forces and \mathbf{x} is a vector of primal variables such as nodal velocities or displacements. In this paper we limit ourself to problems arising in (constrained) nonlinear solid mechanics though the given methods and algorithms might be applicable to other types of nonlinear problems as well.

It is common to apply some variant of Newton's method or some kind of Newton–Krylov method to Eq. (1). With the exception of matrix-free Newton type algorithms, if \mathbf{F} is differentiable at \mathbf{x} , a Jacobian

$$\mathbf{J}(\mathbf{x}) = \left[J_{ij} \right](\mathbf{x}) = \frac{\partial F_i}{\partial x_j}(\mathbf{x}) \quad (2)$$

is formed and used in the Newton– or Newton–Krylov type algorithm [14]. This differentiation can be performed analytically in some cases, or the Jacobian can be formed by a numerical finite difference approximation to the differentiation. Newton type methods in general have very good convergence characteristics but are not guaranteed to converge. The initial estimate \mathbf{x}_0 has to be sufficiently close to the solution and the Jacobian \mathbf{J} has to be a sufficiently good approximation to Eq. (2). Here, we focus on cases where either one or both of these conditions are not met so that Newton's method might not be an appropriate choice.

Besides these Newton type methods, there exist several other solution approaches as e.g. the nonlinear Gauss–Seidel algorithm [6], preconditioned nonlinear conjugate gradient method (nonlinear CG) [2] or nonlinear multigrid schemes [3],[6].

We focus on a variational nonlinear multigrid approach called the full approximation scheme (FAS) that adopts a smoothed aggregation algebraic multigrid method [18],[19],[20] to create prolongation and restriction operators between grids in a purely algebraic manner. No coarse grid discretizations of the underlying problem have to be provided. We use nonlinear CG and/or quasi–Newton methods as nonlinear smoothers and approximate nonlinear solvers on fine and coarse grids.

In addition we discuss the possibility to augment this basic algorithm with an automatically generated Jacobian by applying a block colored finite difference scheme. With this capability inside the nonlinear solver, the underlying application does not have to provide any Jacobian to the multigrid algorithm. The algorithm is therefore suitable to use in applications that merely have functionality to evaluate residuals at a given current solution state as would usually be the case in purely explicit time integration codes.

Underlying theory and algorithms are discussed briefly in this article. The emphasis lies on the usage of the implementation of the proposed algorithm in parallel within the software project *Trilinos* [9] and on the discussion of the several method variants that can be used. We provide two examples comparing several choices.

The implementation makes use of several of *Trilinos*' subpackages, most importantly the algebraic multigrid package *ML* [15] and the nonlinear solver package *NOX* [16]. The presented method is implemented in *ML* and uses its algebraic multigrid hierarchy. Nonlinear smoothing and solution methods as well as the interface to an underlying application code are derived from the *NOX* package.

This article is organized as follows. In Section 2 through Section 6 we present the proposed nonlinear multigrid algorithm and pay special attention to its building blocks. In Section 7 the usage of the method for problems with constraints is discussed and in Section 8 two examples are given. Section 9 serves as a detailed documentation of the implementation of the method and we conclude in Section 10.

2 Smoothed aggregation multigrid (SA)

A multigrid solver tries to approximate the original PDE problem of interest on a hierarchy of grids and uses approximate solutions from coarse grids to accelerate the convergence on the finest grid (which is the one of interest). To do so, a basic – in this case nonlinear – iterative method has to be applied on each grid which effectively smooths out the error associated with the current approximate solution on that grid.

The prolongation \mathbf{P} and restriction \mathbf{R} (operators that transfer solutions, residuals and corrections from coarse to fine grids and vice versa, respectively) are the key ingredient and are determined by the smoothed aggregation method, for details see [18],[19],[20]. The user does not need to supply any hierarchy of coarse discretizations. However, it is advantageous to supply some extra information about the problem, namely the kernel of the fine grid Jacobian neglecting any Dirichlet boundary conditions. In the case of 3D–elasticity, the kernel consists of six vectors describing the rigid body modes of the object of interest neglecting Dirichlet boundary conditions. Computing these rigid body modes is not part of the presented implementation though details on how to compute them from nodal coordinates of the fine grid are given in Section 9.2. The actual fine grid Jacobian is not needed to compute rigid body modes.

The implementation of the presented method also provides the capability to use an adaptive smoothed aggregation procedure (α SA) [4] to identify additional near nullspace modes that are not well captured by an existing multigrid hierarchy. These additional adaptively computed modes are then added to the nullspace and this enriched near–nullspace is used to recompute a multigrid hierarchy with improved approximation and convergence properties.

As the adaptive smoothed aggregation multigrid procedure uses *linear* v–cycles to deter-

mine additional nullspace modes in the setup phase it is inexpensive compared to the actual *nonlinear* iteration to be performed. It is therefore especially suitable to use in the nonlinear setting, which does not always hold when used in *linear* multigrid methods. An example of the adaptive smoothed aggregation approach is given in Section 8.3. The usage with the *Trilinos* implementation is described in Section 9.5.

Systems of equations resulting from finite element or other types of discretizations have a sparse nodal block structure. When the underlying application retains degrees of freedom that are prescribed by Dirichlet boundary condition within the system, the nodal block size is constant throughout the system. However, some applications condense rows and columns associated with Dirichlet boundary condition from the system of equations resulting in a variable block size. In those cases it is beneficial or even necessary to provide *ML* with additional information about the block structure of the problem. *ML* supports variable blocked systems in the generation of the multigrid hierarchy when sufficient information is provided. Building and providing nodal block information as well as choosing the correct input parameters for *ML* is described in Section 9.2.

3 Full approximation scheme (FAS)

The algorithm is described for a nonlinear multigrid V-cycle, which indicates the order that coarse grids are visited and how they contribute to the solution. We use a classical *full approximation scheme* (FAS) [3],[6], that can be shown to reduce to a standard linear V-cycle in the case that the nonlinear function \mathbf{F} in Eq. (1) is in fact linear (see Remark 2).

In this paper sub- and superscripts $()_{(l)}$, $()^{(l)}$ in parenthesis indicate a grid in the multigrid hierarchy with $l = 0$ being the finest grid and L is the number of grids to be used. In the following, the FAS V-cycle is explained as a 2-grid method choosing $L = 2$. Introducing an iterate index k we can rewrite problem (1) on the fine grid as a series of iterates such that

$$\mathbf{F}_{(0)}^k(\mathbf{x}_{(0)}^k) \rightarrow \mathbf{0}. \quad (3)$$

We introduce a nonlinear smoothing method $S_{(l)}^{\nu_1}(\mathbf{F}_{(l)}^k, \mathbf{x}_{(l)}^k)$ to be applied ν_1 times and obtain the presmoothed fine grid iterate

$$\tilde{\mathbf{x}}_{(0)}^k \leftarrow S_{(0)}^{\nu_1}(\mathbf{F}_{(0)}^k, \mathbf{x}_{(0)}^k). \quad (4)$$

Restricting the current solution vector $\tilde{\mathbf{x}}_{(0)}^k$ and the current residual $\mathbf{F}_{(0)}^k(\tilde{\mathbf{x}}_{(0)}^k)$ to the next coarser grid yields

$$\bar{\mathbf{x}}_{(1)} = \mathbf{R}_{(0)}^{(1)} \tilde{\mathbf{x}}_{(0)}^k, \quad \bar{\mathbf{F}}_{(1)} = \mathbf{R}_{(0)}^{(1)} \mathbf{F}_{(0)}^k(\tilde{\mathbf{x}}_{(0)}^k). \quad (5)$$

We omit the iterate index k on all coarse grid variables for clarity.

A modified coarse grid problem

$$\mathbf{F}_{(1)}(\mathbf{x}_{(1)}) - \hat{\mathbf{F}}_{(1)}(\bar{\mathbf{x}}_{(1)}) + \bar{\mathbf{F}}_{(1)} \rightarrow \mathbf{0} \quad (6)$$

is set up, where $\bar{\mathbf{F}}_{(1)}$ is obtained from Eq. (5) and

$$\hat{\mathbf{F}}(\bar{\mathbf{x}}_{(1)}) = \hat{\mathbf{F}}(\mathbf{R}_{(0)}^{(1)} \tilde{\mathbf{x}}_{(0)}^k). \quad (7)$$

$\bar{\mathbf{F}}_{(1)}$, $\hat{\mathbf{F}}_{(1)}$ are fixed quantities throughout the nonlinear smoothing iteration $S_{(1)}^{\nu_1}$ applied to Eq. (6)

$$\tilde{\mathbf{x}}_{(1)} \leftarrow S_{(1)}^{\nu_1}(\mathbf{F}_{(1)} - \hat{\mathbf{F}}_{(1)} + \bar{\mathbf{F}}_{(1)}, \mathbf{x}_{(1)}). \quad (8)$$

The nonlinear function $\mathbf{F}_{(l)}$ is evaluated in a variational way on all coarse grids $l > 0$. Specifically,

$$\mathbf{F}_{(l)}(\mathbf{x}_{(l)}) = \mathbf{R}_{(0)}^{(l)} \mathbf{F}_{(0)}(\mathbf{P}_{(l)}^{(0)} \mathbf{x}_{(l)}) \quad , \quad l > 0, \quad (9)$$

where $\mathbf{P}_{(k)}^{(j)}$ prolongates from level k to level j and $\mathbf{R}_{(j)}^{(k)}$ restricts from level j to level k . Specifically

$$\mathbf{P}_{(l)}^{(0)} = \mathbf{P}_{(1)}^{(0)} \mathbf{P}_{(2)}^{(1)} \dots \mathbf{P}_{(l)}^{(l-1)} \quad , \quad (10)$$

$$\mathbf{R}_{(0)}^{(l)} = \mathbf{P}_{(l)}^{(0)T} \quad , \quad (11)$$

and $\mathbf{R}_{(l-1)}^{(l)T} = \mathbf{P}_{(l)}^{(l-1)}$ correspond to the smoothed aggregation multigrid hierarchy discussed in Section 2.

Remark 1: Operators $\mathbf{P}_{(l)}^{(0)}$, $\mathbf{R}_{(0)}^{(l)}$ are never explicitly formed. Instead, Eq. (10) is applied.

Given the nonlinear smoothed coarse grid iterate $\tilde{\mathbf{x}}_{(1)}$, a correction is calculated and added to the current fine grid solution using the interpolation operator:

$$\tilde{\mathbf{x}}_{(0)}^k \leftarrow \tilde{\mathbf{x}}_{(0)}^k + \mathbf{P}_{(1)}^{(0)}(\tilde{\mathbf{x}}_{(1)} - \bar{\mathbf{x}}_{(1)}). \quad (12)$$

Applying a postsmoothing step

$$\mathbf{x}_{(0)}^{k+1} \leftarrow S_{(0)}^{\nu_2}(\mathbf{F}_{(0)}^k, \tilde{\mathbf{x}}_{(0)}^k). \quad (13)$$

ν_2 times finalizes the V-cycle. Fig. 1 gives the complete FAS solver scheme for the general case $L \geq 2$. Therein, $\bar{\mathbf{F}}_{(l)}$ denotes the residual restricted to level l which remains unaltered throughout the pre- and postsmoothing steps. The number of smoothing steps ν_1 and ν_2 can be chosen independently on each grid. Choosing $\nu_1 = \nu_2$ on each grid results in a symmetric

V-cycle which is normally recommended for symmetric problems in the linear case. Choosing $\nu_1 \neq \nu_2$ results in a nonsymmetric multigrid operator which can be competitive with respect to performance in the nonlinear case.

```

FAS_Vcycle (  $\bar{\mathbf{F}}_{(l)}$ ,  $\bar{\mathbf{x}}_{(l)}$ ,  $l$  )
  if  $l = 0$ 
    Presmooth  $\mathbf{x}_{(l)} \leftarrow S_{(l)}^{\nu_1}(\mathbf{F}_{(l)}, \mathbf{x}_{(l)})$ 
  else
     $\hat{\mathbf{F}}_{(l)} \leftarrow \mathbf{F}_{(l)}(\bar{\mathbf{x}}_{(l)})$ 
    Presmooth  $\mathbf{x}_{(l)} \leftarrow S_{(l)}^{\nu_1}(\mathbf{F}_{(l)} - \hat{\mathbf{F}}_{(l)} + \bar{\mathbf{F}}_{(l)}, \mathbf{x}_{(l)})$ 
  endif
  if  $l < L - 1$ 
     $\mathbf{x}_{(l+1)} = \mathbf{R}_{(l)}^{(l+1)} \mathbf{x}_{(l)}$ 
     $\mathbf{F}_{(l+1)} = \mathbf{R}_{(l)}^{(l+1)} \mathbf{F}_{(l)}(\mathbf{x}_{(l)})$ 
    FAS_Vcycle (  $\mathbf{F}_{(l+1)}$ ,  $\mathbf{x}_{(l+1)}$ ,  $l + 1$  )
     $\mathbf{x}_{(l)} \leftarrow \mathbf{x}_{(l)} + \mathbf{P}_{(l+1)}^{(l)} \mathbf{x}_{(l+1)}$ 
  endif
  if  $l = 0$ 
    Postsmooth  $\mathbf{x}_{(l)} \leftarrow S_{(l)}^{\nu_1}(\mathbf{F}_{(l)}, \mathbf{x}_{(l)})$ 
  else
    Postsmooth  $\mathbf{x}_{(l)} \leftarrow S_{(l)}^{\nu_1}(\mathbf{F}_{(l)} - \hat{\mathbf{F}}_{(l)} + \bar{\mathbf{F}}_{(l)}, \mathbf{x}_{(l)})$ 
     $\bar{\mathbf{x}}_{(l)} \leftarrow \bar{\mathbf{x}}_{(l)} - \mathbf{x}_{(l)}$ 
  endif
return

```

Figure 1: FAS V-cycle as a solver

The full approximation scheme V-cycle can also be easily formulated as a preconditioner to some outer nonlinear iteration, e.g. nonlinear CG. The V-cycle as a preconditioner is given in Fig. 2. The *ML* implementation supports both versions.

Remark 2: When $\mathbf{F}_{(0)}$ is linear and an exact coarse grid solve is used the algorithm corresponds to the usual multigrid scheme. In particular:

$$\mathbf{F}_{(0)} = \mathbf{b}_{(0)} - \mathbf{A}_{(0)}\mathbf{x}_{(0)} \quad , \quad \mathbf{F}_{(1)} = \mathbf{b}_{(1)} - \mathbf{A}_{(1)}\mathbf{x}_{(1)} . \quad (14)$$

Substituting Eq. (14) into (6) we get

$$\mathbf{b}_{(1)} - \mathbf{A}_{(1)} \mathbf{x}_{(1)} = \mathbf{b}_{(1)} - \mathbf{A}_{(1)} \mathbf{R}_{(0)}^{(1)} \mathbf{x}_{(0)} - \mathbf{R}_{(0)}^{(1)} (\mathbf{b}_{(0)} - \mathbf{A}_{(0)} \mathbf{x}_{(0)}) \quad (15)$$

$$\mathbf{A}_{(1)} \mathbf{x}_{(1)} = \mathbf{A}_{(1)} \mathbf{R}_{(0)}^{(1)} \mathbf{x}_{(0)} + \mathbf{R}_{(0)}^{(1)} (\mathbf{b}_{(0)} - \mathbf{A}_{(0)} \mathbf{x}_{(0)}) \quad (16)$$

Multiplying by an exact coarse grid solve $\mathbf{A}_{(1)}^{-1}$ from the right yields

$$\mathbf{x}_{(1)} = \mathbf{R}_{(0)}^{(1)} \mathbf{x}_{(0)} + \mathbf{A}_{(1)}^{-1} \mathbf{R}_{(0)}^{(1)} (\mathbf{b}_{(0)} - \mathbf{A}_{(0)} \mathbf{x}_{(0)}). \quad (17)$$

Substituting this into Eq. (12) yields the usual linear multigrid coarse level correction

$$\mathbf{x}_{(0)} \leftarrow \mathbf{x}_{(0)} + \mathbf{P}_{(1)}^{(0)} \mathbf{A}_{(1)}^{-1} (\mathbf{R}_{(0)}^{(1)} (\mathbf{b}_{(0)} - \mathbf{A}_{(0)} \mathbf{x}_{(0)})). \quad (18)$$

The proposed method is matrix-free in the sense that the application does not need to supply a Jacobian. A Jacobian or some approximation to it can be constructed internally by finite differencing and used in the smoother or for the coarse grid solve. It is though always beneficial if the underlying application is capable of providing a Jacobian. Analytical Jacobians are of better quality than obtained by finite differencing and the finite differencing process is a high percentage of the overall setup cost of the method.

FAS_Vcycle_p ($\bar{\mathbf{F}}_{(l)}$, $\bar{\mathbf{x}}_{(l)}$, l)

$$\hat{\mathbf{F}}_{(l)} \leftarrow \mathbf{F}_{(l)}(\bar{\mathbf{x}}_{(l)})$$

$$\text{Presmooth } \mathbf{x}_{(l)} \leftarrow S_{(l)}^{v_1} (\mathbf{F}_{(l)} - \hat{\mathbf{F}}_{(l)} + \bar{\mathbf{F}}_{(l)}, \mathbf{x}_{(l)})$$

if $l < L-1$

$$\mathbf{x}_{(l+1)} = \mathbf{R}_{(l)}^{(l+1)} \mathbf{x}_{(l)}$$

$$\mathbf{F}_{(l+1)} = \mathbf{R}_{(l)}^{(l+1)} \mathbf{F}_{(l)}(\mathbf{x}_{(l)})$$

$$\text{FAS_Vcycle_p} (\mathbf{F}_{(l+1)}, \mathbf{x}_{(l+1)}, l+1)$$

$$\mathbf{x}_{(l)} \leftarrow \mathbf{x}_{(l)} + \mathbf{P}_{(l+1)}^{(l)} \mathbf{x}_{(l+1)}$$

endif

$$\text{Postsmooth } \mathbf{x}_{(l)} \leftarrow S_{(l)}^{v_1} (\mathbf{F}_{(l)} - \hat{\mathbf{F}}_{(l)} + \bar{\mathbf{F}}_{(l)}, \mathbf{x}_{(l)})$$

$$\bar{\mathbf{x}}_{(l)} \leftarrow \bar{\mathbf{x}}_{(l)} - \mathbf{x}_{(l)}$$

return

Figure 2: FAS V-cycle as a preconditioner

4 Nonlinear conjugate gradient method (nonlinear CG) as a smoother and solver

The preconditioned nonlinear CG method [2],[7],[8] is adopted as one choice of a nonlinear smoother/solver in the hierarchy of grids. It can also be used as an outer nonlinear Krylov meth-

od to which the presented nonlinear algebraic multigrid is then applied as a nonlinear preconditioner. Given the nonlinear problem $\mathbf{F}_{(l)}^k(\mathbf{x}_{(l)}^k)$ from Eq. (1) on some grid (l) with $k = 0, \dots, N^{iter}$ as iterate index a search direction

$$\mathbf{s}^{k+1} = \mathbf{M}^{-1} \mathbf{F}^k, \quad k = 0, \quad (19)$$

$$\mathbf{s}^{k+1} = \mathbf{M}^{-1} \mathbf{F}^k + \beta^k \mathbf{s}^k, \quad k > 0 \quad (20)$$

is computed omitting the grid identifier (l) for clarity, where \mathbf{M} is a *linear* preconditioner and β^k results from the so called Polak–Ribière formula

$$\beta^k = \frac{\mathbf{F}^{kT} \mathbf{M}^{-1} (\mathbf{F}^k - \mathbf{F}^{k-1})}{\mathbf{F}^{k-1T} \mathbf{M}^{-1} \mathbf{F}^{k-1}}. \quad (21)$$

In the case $\beta^k < 0$ the iteration is restarted using Eq. (19). A new iterate

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \mathbf{s}^{k+1} \quad (22)$$

is computed using the line search parameter

$$\alpha^k = \frac{\mathbf{F}^{kT} \mathbf{s}^k}{\mathbf{F}(\mathbf{x}^k + \mathbf{s}^{k+1})^T \mathbf{s}^k - \mathbf{F}^{kT} \mathbf{s}^k}. \quad (23)$$

The iteration terminates when a user provided convergence criteria $\|\mathbf{F}^k\|_2 < \epsilon$ is met or a prescribed maximum number of iterations is reached.

Except for the *linear* preconditioner \mathbf{M} to the nonlinear CG, no Jacobian is used to generate a search direction as would be the case in linear CG. The implementation of the method offers a variety of choices for \mathbf{M} , e.g. damped Jacobi, domain decomposition symmetric Gauss–Seidel, Chebychev polynomials and a LU–factorization, see Section 9.5. All of them except for the Jacobi preconditioner need a Jacobian for construction. If one sweep of damped Jacobi preconditioning is chosen by the user, just the main diagonal of the Jacobian is necessary and can be efficiently constructed using finite differencing.

Though nonlinear CG has been proven to be inferior to Newton’s method with respect to convergence rates in [2], it is more reliable if the approximation to the Jacobian is not of high quality and/or the initial guess is far from the solution. It also is of low computational cost and might actually be very efficient depending on the efficiency of the underlying application in evaluating the residual.

Remark 3: *Code that has been designed for explicit time integration usually does not have a supporting infrastructure to compute and assemble Jacobians. On the other hand, such applications usually compute residual vectors in a very efficient way. Thus, nonlinear CG (even though it generally takes more iterations than some Newton type method) might actually be the nonlinear smoother of choice. Our numerical studies also indicate that it seems to be less sensitive to poor quality approximations to a Jacobian than the quasi–Newton method, as the Jacobian is used in the construction of the linear preconditioner to the nonlinear CG only.*

5 Quasi-Newton method as a smoother and solver

The implementation of the proposed variational multigrid offers a choice of two types of quasi-Newton method. The first is a simple modified Newton method (with a guaranteed linear convergence rate), where the Jacobian \mathbf{J}^0 is computed or provided in the setup phase of the multigrid and is used throughout the iteration on some grid (l):

$$\Delta \mathbf{x}^k = -(\mathbf{J}^0)^{-1} \mathbf{F}(\mathbf{x}^k) \quad , \quad \mathbf{x}^{k+1} = \mathbf{x}^k + \Delta \mathbf{x}^k \quad , \quad k \leftarrow k + 1 \quad (24)$$

The second is matrix-free Newton-Krylov, where the matrix-vector product of the Krylov solver is approximated by

$$\mathbf{J}^k \mathbf{y} \approx \frac{\mathbf{F}(\mathbf{x}^k + \delta \mathbf{y}) - \mathbf{F}(\mathbf{x}^k)}{\delta} \quad , \quad (25)$$

and $\delta \ll 1$ is a perturbation parameter.

As a linear solver inside the Newton iteration a preconditioned linear Krylov method is used. As preconditioner to the Krylov solver, one of the methods mentioned in Section 4 can be chosen. Additionally, the number of Krylov iterations and Newton iterations can be limited separately on each grid to allow for incomplete solves, see also Section 9.5. *NOX* provides the implementation of Newton's method, using the *AztecOO* [10] package's implementation of parallel preconditioned Krylov iterative methods.

6 Block colored finite differencing of Jacobian operators

As some applications might not provide a Jacobian matrix, the construction of a Jacobian on some grid (l) can be performed using a parallel block colored finite difference scheme. The minimum requirement to the underlying application is therefore to provide a graph (sparsity pattern) of the problem on the fine grid, see also Section 9.5.

Scalar entries of the tangent Jacobian operator are approximated by a secant. So called forward differencing evaluates

$$\mathbf{J}_{ij} = \frac{\partial \mathbf{F}_i}{\partial x_j} \approx \frac{\mathbf{F}_i(\mathbf{x} + \delta \mathbf{e}_j) - \mathbf{F}_i(\mathbf{x})}{\delta} \quad , \quad \delta = \alpha |x_j| + \beta \quad , \quad (26)$$

where \mathbf{e}_j is the j th unit vector and δ is a scalar perturbation value computed from user chosen parameters α and β . Forward finite differencing needs $N + 1$ evaluations of the residual.

Central finite differencing evaluates

$$\mathbf{J}_{ij} = \frac{\partial \mathbf{F}_i}{\partial x_j} \approx \frac{\mathbf{F}_i(\mathbf{x} + \delta \mathbf{e}_j) - \mathbf{F}_i(\mathbf{x} - \delta \mathbf{e}_j)}{2\delta} \quad , \quad (27)$$

and provides second order spatial accuracy at the cost of $2N + 1$ evaluations of the residual vector. Both methods cannot be used in their original form as the computational cost is immense and scales quadratically with respect to problem size. Therefore, the problem graph is colored using a parallel distance-2 coloring scheme such that every colored node of the graph does not share a neighbor with any other node of the same color. A graphical illustration of a distance-2 coloring of a structured graph is given in Fig. 3.

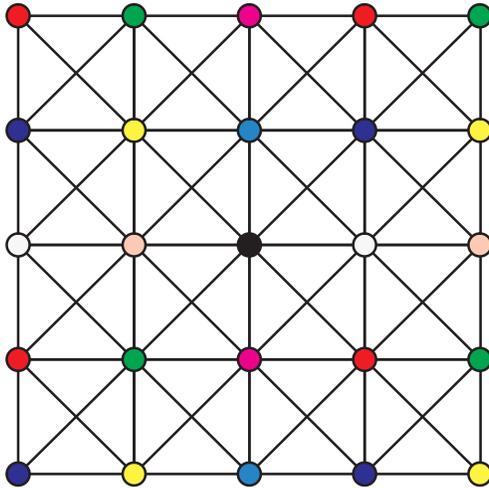


Figure 3: Distance-2 graph coloring

All entries sharing a color can then be evaluated at the same time reducing the number of residual evaluations to the number of colors. The distance-2 coloring process scales as $\mathcal{O}(Nb^2)$, where b is the maximum bandwidth of the graph and N is the problem size. The coloring is performed exploiting the nodal block structure of the problem resulting from the finite element nature of the problem. By building a nodal block graph, which is smaller in N as well as in b by the factor n , where n is the number of degrees of freedom per node, the coloring cost can be reduced to $\mathcal{O}\left(\frac{Nb^2}{n^3}\right)$. This proved to be affordable in all tested applications. Note that while the actual parallel coloring is performed by the *Trilinos* subpackage *EpetraExt* [12], the nodal block collapsed coloring wrapper is currently implemented in *ML*.

7 Nonlinear systems of equations with constraints

The nonlinear multigrid method can also be used to solve nonlinear systems of equations with constraints

$$\mathbf{F}(\mathbf{x}) + \mathbf{C}\boldsymbol{\lambda} = \mathbf{0}, \quad (28)$$

subject to

$$\mathbf{C}^T \mathbf{x} = \mathbf{0}, \quad (29)$$

where $\boldsymbol{\lambda}$ is a set of Lagrange multipliers and \mathbf{C} is a matrix representation of constraint equations. Such constraints can arise from e.g. contact formulations, mesh-tying and periodic boundary

conditions.

The solver is designed to operate on x only expecting the underlying application to perform necessary constraint enforcement. It has to be guaranteed that the current iterate and initial guess satisfy constraints Eq. (29) and the residual is evaluated according to Eq. (28). Note that convergence might deteriorate when C is nonlinear.

Special care should be taken in the construction of the Jacobian matrix used to build the multigrid hierarchy and the smoothing operators. The Jacobian (gradient of F) should at least satisfy the constraints approximately. This can be achieved by either using a penalty approach for the constraints or by using the colored finite differencing process described in Section 6, where Eqns. (28) and (29) are applied to the probe vector and residual, respectively.

In the latter case of colored finite differencing for an appropriate Jacobian, a graph containing all potential Jacobian entries for constraints has to be supplied and is used in the coloring and finite differencing process.

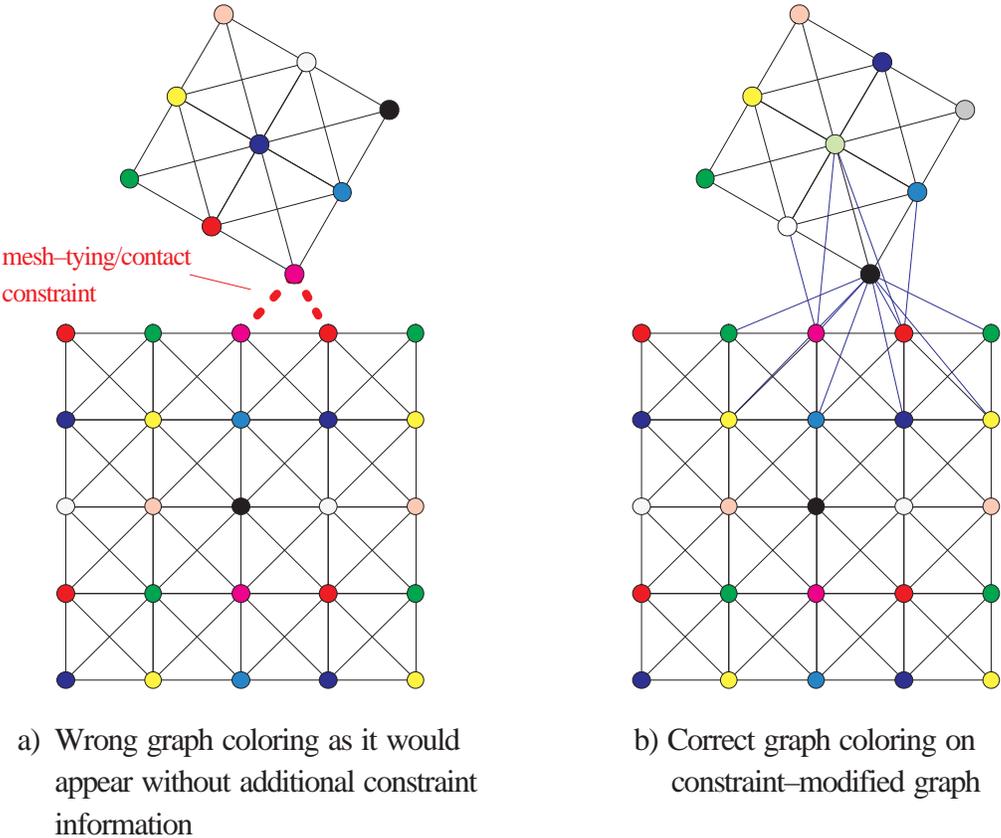


Figure 4: Distance-2 graph coloring of constraint-modified graph

It is requested by the implementation of the method through the `getModifiedGraph()` method described in Section 9.2. The way in which the graph has to be modified depends on the type of the constraints and is not prescribed by the solver.

By way of example we describe the graph modification when strong local contact or mesh tying constraints are considered. Fig. 4a shows the graph of a problem where one slave node is constrained to two master nodes. As the multigrid algorithm operates on primal degrees of freedom only, the graph of the problem does not contain any information on this constraint. Therefore, the distance-2 coloring process will not consider the constraint and construction of a finite difference Jacobian using this coloring will lead to the wrong result.

The application therefore has to supply a modified graph of the problem as given in Fig. 4b. This graph coloring will result in a Jacobian where constraints are handled correctly. In case of non-local or variational constraint formulations such as e.g. Mortar methods [21] the above described correct graph modification might be elaborate or impossible to form. In such cases it is better to rely on the specific properties of the constraint formulation to form a Jacobian that is definite.

In the nonlinear multigrid V-cycle described in Section 3 the residual is evaluated in a variational form Eq. (9). Given a current iterate $\mathbf{x}_{(0)}^k$ and matching residual $\mathbf{F}_{(0)}^k$ let us assume the underlying application has some method to enforce constraints $\mathcal{C}_{(0)} : \mathbf{x}_{(0)}^k \rightarrow \tilde{\mathbf{x}}_{(0)}^k$, such that $\tilde{\mathbf{x}}_{(0)}^k$ satisfies Eq. (29) and $\tilde{\mathbf{F}}_{(0)}^k$ matching $\tilde{\mathbf{x}}_{(0)}^k$ is computed from Eq. (28). Then, constraints on coarse grids $l > 0$ are enforced in a variational way

$$\tilde{\mathbf{F}}_{(l)}(\mathbf{x}_{(l)}) = \mathbf{R}_{(0)}^{(l)} \tilde{\mathbf{F}}_{(0)} \left(\mathcal{C}_{(0)} \mathbf{P}_{(l)}^{(0)} \mathbf{x}_{(l)} \right), \quad (30)$$

$$\tilde{\mathbf{x}}_{(l)} = \mathbf{R}_{(0)}^{(l)} \mathcal{C} \mathbf{P}_{(l)}^{(0)} \mathbf{x} \quad (31)$$

in each nonlinear smoothing step, where $\mathbf{P}_{(l)}^{(0)}$, $\mathbf{R}_{(0)}^{(l)}$ were defined in Eqns. (10) and (11).

As prolongation and restriction operators obtained from the smoothed aggregation multigrid method do not conserve the scaling of the iterate in Eq. (31) due to

$$\mathbf{R}_{(0)}^{(l)} \mathbf{P}_{(l)}^{(0)} \neq \mathbf{I}_{(l)}, \quad (32)$$

the scaling of $\tilde{\mathbf{x}}_{(l)}$ is suboptimal. This issue is subject to current research and improvement.

Constrained enforcement has to be explicitly turned on in the parameters for the nonlinear preconditioner as described in Section 9.5.

8 Examples

8.1 One dimensional example

This simple example discretizes and solves

$$\frac{\partial^2 u}{\partial x^2} - 10^3 u^2 = 0 \quad , \quad \Omega = [0, 1] \quad , \quad u(x = 0) = 1 \quad , \quad u(x = 1) = 0 \quad (33)$$

using linear finite elements. It is also provided as a user example with the distribution of *Trilinos/ML*, see Section 9.6.

We fix the size of the generated coarsest grid to be 3000 equations and the coarsening rate to be 1 : 3 for each level. This leads to an increase in the number of coarse grids as the problem size is increased by refinement. We use colored finite differencing to obtain a fine grid Jacobian matrix from which a smoothed aggregation multigrid hierarchy is generated. We denote this set of choices as ‘Version I’ in Fig. 5. As nonlinear smoothers, we select a polynomial smoother[1] preconditioned nonlinear CG where the polynomial order is chosen as 4 on all grids. On the coarsest grid, we use nonlinear CG preconditioned by a LU factorization of the variational coarsest level Jacobian. We apply a nonsymmetric V-cycle skipping all presmoothing steps on all grids to avoid the presmoothing residual evaluations, applying 6 iterations on the coarsest grid, 2 postsmoothing steps on every intermediate grid and 3 postsmoothing steps on the finest grid, respectively. As convergence criteria, an absolute residual norm of $1.0e-07$ is chosen.

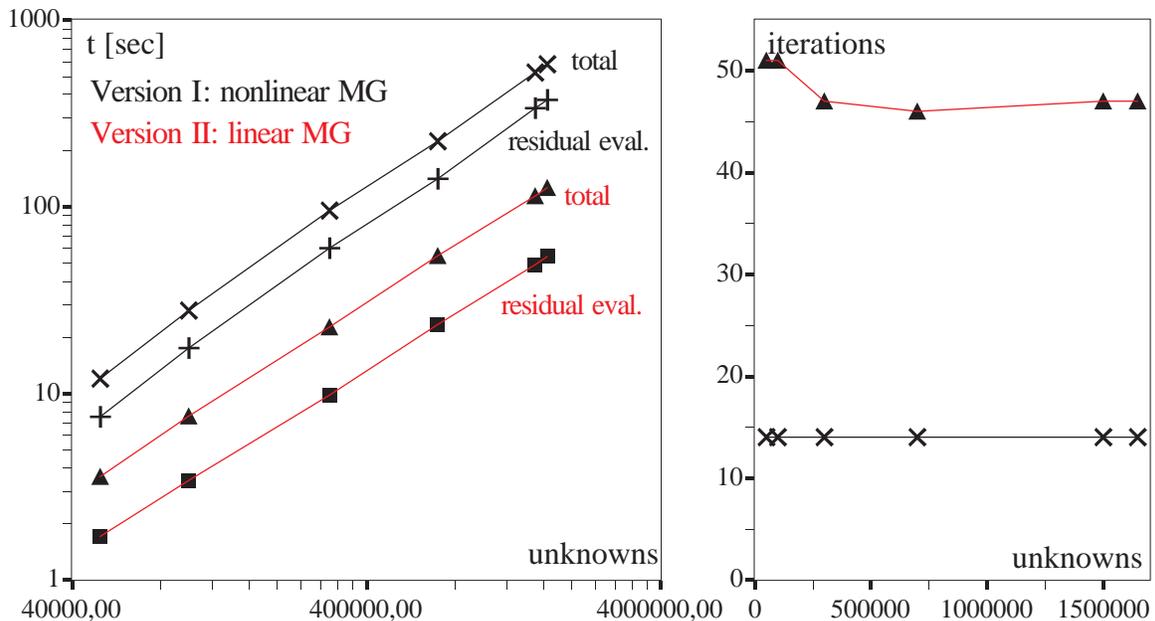


Figure 5: Solution times

A second variant denoted ‘Version II’ in Fig. 5 uses a *linear* multigrid operator as a preconditioner to an outer nonlinear CG. A polynomial smoother of degree 3 is used on all grids except for the coarse grid where a direct solve is applied. The Jacobian is generated on the fine grid using finite differencing and a smoothed aggregation multigrid hierarchy is constructed as before.

Fig. 5 provides total solution times including setup and time spent in the residual evaluation as well as number of outer nonlinear CG iterations taken for various refinements. All times were obtained on a dual Xeon 3.6 GHz machine.

It can be seen that in this case using a standard linear preconditioner is superior over the nonlinear variant. For both versions, the number of nonlinear CG iterations remains constant while increasing the problem size. It is not so easy to see in this picture but the overall time for the nonlinear multigrid (Version I) increases faster than the time spent in the residual evaluation. This is due to the variational evaluation of the nonlinear function on coarse grids, see Eqns. (9) and (10). As the problem size increases and more coarse grids are added, the transfer of the current iterate on some coarse grid to the fine grid and the transfer of the residual vector back to the coarse grid gets more expensive as more intermediate grids exist. It is therefore recommended to use as few grids as possible by e.g. applying aggressive coarsening strategies. As will be shown in the second example in Section 8.2, the nonlinear algorithm is more attractive for problems with severe nonlinearities and ill-conditioned Jacobian matrices.

8.2 Three dimensional example

As a second example, a thin walled half sphere is studied which is discretized using a nonlinear large deformation hybrid three-dimensional shell formulation [5].

The sphere is loaded by an internal hydrostatic pressure load (which also is nonlinear) in a transient analysis. Snapshots of the deformation and simulation parameters are given in Fig. 6. Due to the high radius to thickness ratio and the fact that the thickness change of the sphere wall is taken into account by the shell formulation, the Jacobian operator shows severe ill-conditioning.

We study three versions of nonlinear and linear multigrid preconditioners. In Fig. 7, the versions are described and solution times including the setup phase and iteration numbers are given for each nonlinear solve.

Versions II and III which are the nonlinear multigrid V-cycles perform best with respect to time in this example. Though version I, which is the linear multigrid preconditioner, performs competitively at the beginning of the simulation, its number of iterations starts increasing drastically around time step 140 as soon as the nonlinearities get more severe. The number of iterations of versions II and III remain low up to the end of the simulation time. The significant increase in the number of iterations at the end of the simulation is due to the appearance

of dynamic buckling of the structure which should make an adjustment of the time step size necessary at that time.

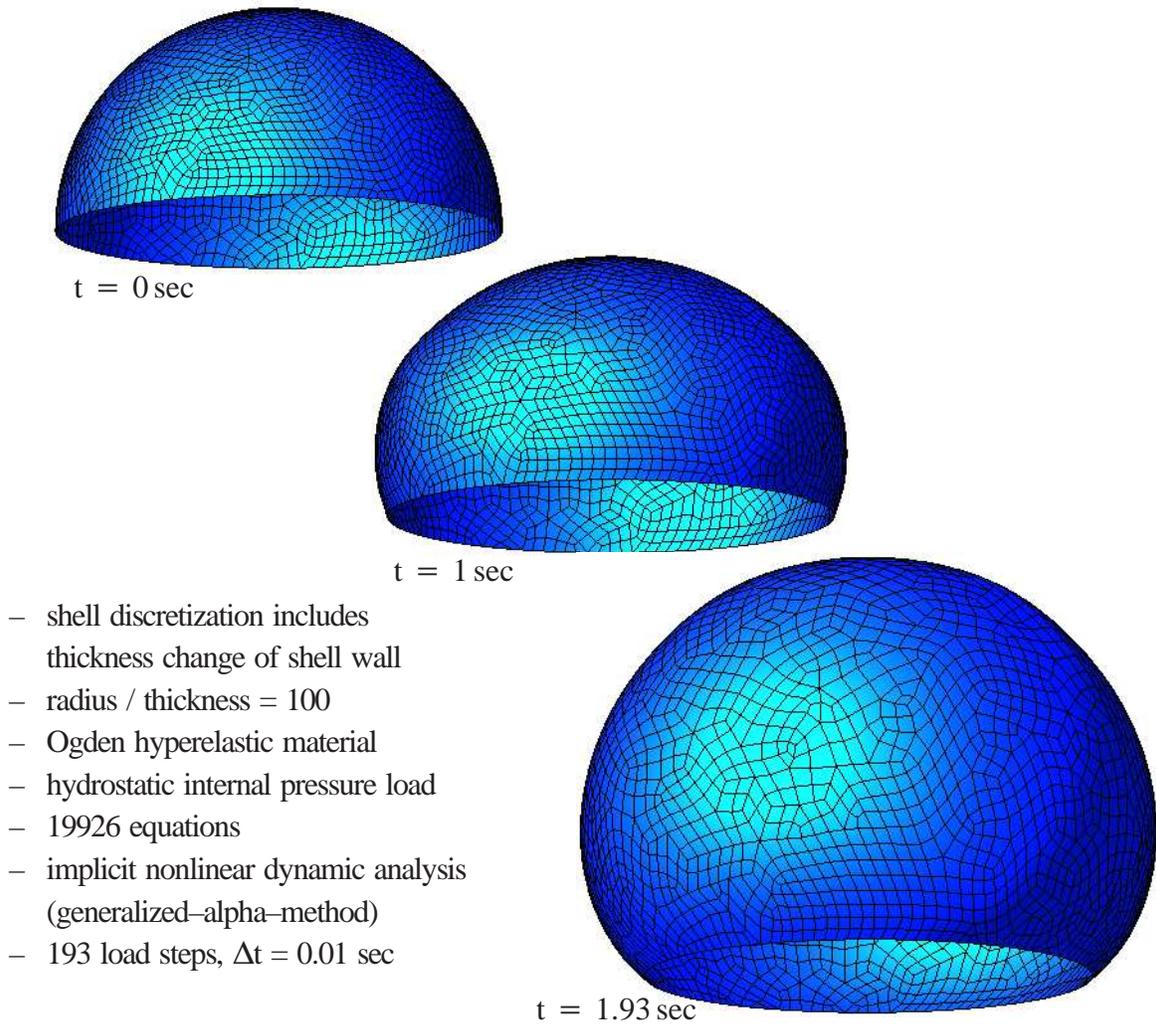
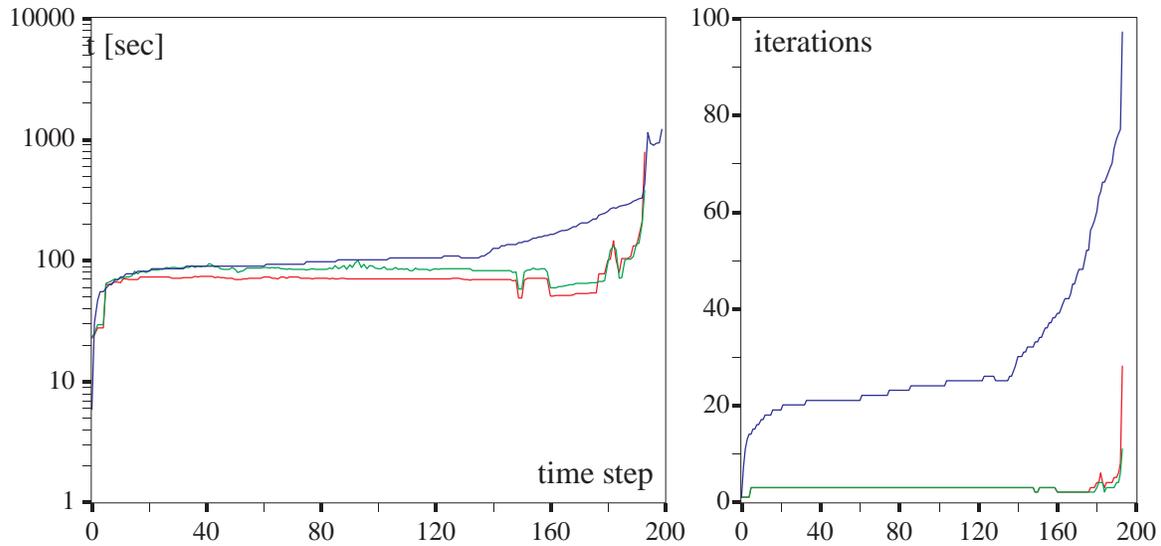


Figure 6: Half sphere under hydrostatic pressure

**Version I**

- 3 grid linear MG preconditioner
- order 4 poly. smoother on fine grid
- 3 sweeps symmetric Gauss–Seidel on medium grid
- LU factorization on coarse grid

Version II

- 3 grid nonlinear MG preconditioner
- preconditioned nonlinear CG smoother on all grids
- order 4 poly. smoother on fine grid
- 3 sweeps DD–symm. Gauss–Seidel on medium grid
- LU factorization on coarse grid

Version III

- 3 grid nonlinear MG preconditioner
- 2 step quasi–Newton method on all grids
- preconditioned CG as linear solver
- order 4 poly. smoother on fine grid
- 3 sweeps DD–symm. Gauss–Seidel on medium grid
- LU factorization on coarse grid

Figure 7: Solution times

8.3 Three dimensional example with adaptive smoothed aggregation

We consider the example from Section 8.2 but significantly reduce the mass term in the non-linear system to increase ill-conditioning even further. This leads to an increase in iteration numbers and overall solution time compared to the example in Section 8.2. Version III of the nonlinear multigrid algorithm described in Fig. 7 is applied with and without the adaptive smoothed aggregation setup procedure [4] described in Section 2.

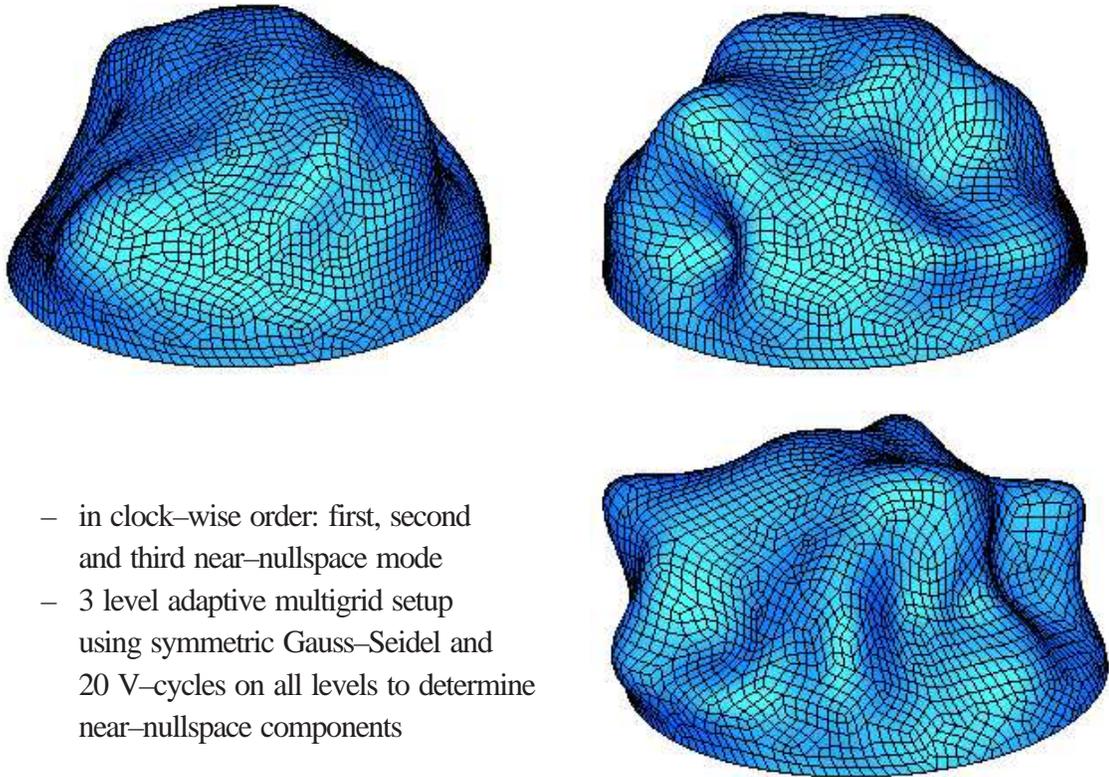


Figure 8: Near-nullspace modes from adaptive smoothed aggregation setup

An initial smoothed aggregation multigrid preconditioner is constructed based on the 6 rigid body modes of the structure and the adaptive smoothed aggregation procedure is applied to compute an additional 3 near-nullspace modes visualized in Fig. 8 that are not well-captured by the existing multigrid preconditioner. The initial 6 rigid body modes together with 3 adaptively computed near-nullspace modes are then incorporated in a refined multigrid hierarchy which is then used in the nonlinear multigrid iteration described in Section 3.

Iteration numbers and solution times that include all setup costs are given in Fig. 9 for a simulation applying 123 time steps.

In Fig. 9 the overall solution time benefits from the adaptive setup even though there is a significantly higher setup cost for the adaptive multigrid hierarchy in each time step and a slightly increased cost for the application of one nonlinear V-cycle. The peak that can be seen in Fig. 9 at time step 65 is due to geometric buckling phenomena which drives the determinant of the Jacobian close to zero thus significantly increasing the condition number of the problem.

Remark 4: *Efficiency can be increased even further if the once computed near-nullspace modes are reused in several consecutive solves as the setup procedure of the adaptive smoothed aggregation method contributes a major component to the overall solution time. This approach was not used here to demonstrate competitiveness of the adaptive method in each individual solve.*

It can also be expected that the benefit from the adaptive procedure is even higher in the case of more complex models with e.g. jumps in material coefficients, where algebraically smooth solution components are less well captured by the standard smoothed aggregation multigrid approach.

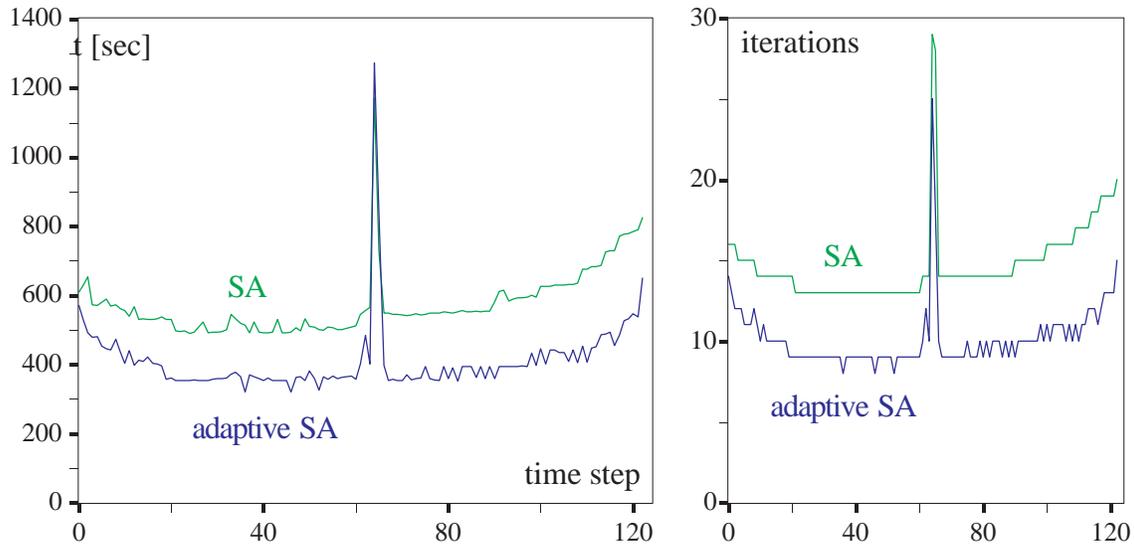


Figure 9: Solution times with adaptive smoothed aggregation setup

9 Implementation documentation

9.1 Availability and configuration of *Trilinos* and *ML*

The proposed algorithm is part of the *ML* [15] package within the *Trilinos* [9] framework. It is contained in the *Trilinos* developer version and in *Trilinos* 6.0 and later releases. Note that this report refers to the *Trilinos* 6.0 version of the code.

```

--enable-ml          --with-ml_metis      --with-ml_nox
--enable-nox         --enable-nox-epetra   --enable-prerelease
--enable-epetra     --enable-epetraext
--enable-teuchos
--enable-aztecoo
--enable-amesos
--with-ldflags="-L/<METIS_PATH>"
--with-incdirs="-I/<METIS_PATH>/Lib"
--with-libs="-lmetis"

```

Figure 10: Configuring *Trilinos* for use of nonlinear MG

As the method makes use of several of *Trilinos*' subpackages such as *ML*, *NOX*, *Epetra* and *EpetraExt*, *Trilinos* has to be configured such that all necessary subpackages are present. To

do so, the configure options given in Fig. 10 should be included in the *Trilinos* configuration. For general installation and usage instructions for *Trilinos* and *ML*, see [9] and [15]. In some choices of aggregation schemes, *ML* makes use of the third party library *Metis* [13] not contained in the *Trilinos* distribution. The user has to provide the *Metis* library when configuring and compiling *Trilinos*, see Fig. 10.

9.2 The application interface

The interface between an underlying nonlinear application and the nonlinear multigrid preconditioner or solver is entirely contained in a virtual class called `ML_NOX::ML_Nox_Fineinterface` contained in the file `Trilinos/packages/ml/src/NonlinML/ml_nox_fineinterface.H`.

The user has to provide an implementation of this virtual class. Data from the solver such as e.g. the current solution iterate will be passed to the underlying application through this interface and the application has to provide information such as a residual vector or a graph of the problem to the solver. The virtual class `ML_NOX::ML_Nox_Fineinterface` itself inherits from three different types of *NOX*'s interface classes.

Remark 5: *It shall be stressed that all data passed to and from the solver through this interface refers to the finest grid which is the one the user is interested in solving. No data associated with any coarse grids nor a coarse grid interface need to be provided as data is passed to coarse grids through automatically generated coarse grid interfaces that obtain data from the described fine grid interface. Most data is passed as Epetra distributed objects. Please refer to the Epetra user manual [11] for a detailed description.*

In the following, a brief description of the methods in `ML_NOX::ML_Nox_Fineinterface` is given. Pure virtual methods have to be implemented by the user's derived class.

```
virtual bool computeF( const Epetra_Vector& x, Epetra_Vector& FVec,  
                      FillType flag) = 0;
```

The solver provides the current iterate (solution vector) in `x` and expects the underlying application to reevaluate the residual vector and store it in `FVec`.

```
virtual bool computeJacobian(const Epetra_Vector& x) = 0;
```

The solver provides the current iterate (solution vector) in `x` and expects the application to reevaluate the Jacobian matrix and store it so it can be accessed by the method `getJacobian()` described below. This method will only be used in cases where the user specified usage of an application provided Jacobian in the input options, see Section 9.5.

```
virtual Epetra_CrsMatrix* getJacobian() = 0;
```

The method returns a pointer to the Jacobian evaluated at the most recent call to `computeJacobian()` described above. This method will only be used in cases where the user specified usage of an application provided Jacobian in the input options.

```
virtual bool computePreconditioner(const Epetra_Vector& x) = 0;
```

The method is currently not used by the algorithm. It has to be implemented though (e.g. with an error message) as it is derived from an underlying *NOX* virtual class.

```
virtual const Epetra_CrsGraph* getGraph() = 0;
```

The method returns the graph of the problem. This could either be the graph underlying the Jacobian matrix if present or, in the case the application does not provide a Jacobian, it has to be a graph instance computed and stored for this purpose by the underlying application.

```
virtual const Epetra_CrsGraph* getModifiedGraph() = 0;
```

The method returns the modified graph of the constrained problem as described in Section 7.

```
virtual const Epetra_Map& getMap() = 0;
```

The method returns the *Epetra* map associated with the solution vector. It is crucial that this map is pointwise identical to the map of the residual vector, the pointwise row map of the Jacobian and the pointwise row map of the supplied graph. In parallel, any map might be specified though the performance of the algorithm benefits from well chosen distributions of the unknowns as can be generated using a partitioning library such as e.g. *Metis* [13].

```
virtual double* Get_Nullspace( const int nummyrows, const int numpde,  
                              const int dim_nullsp) = 0;
```

The method returns vectors representing an approximation to the nullspace of the nonlinear PDE operator. In case of elasticity problems, these vectors contain the six rigid body modes of the discrete domain on the fine grid. The rigid body modes can be easily computed using nodal coordinates of the fine discretization, for details see [15] and the example in Fig. 11. They are used in the construction of the algebraic prolongation and restriction operators and play a crucial role to the performance of the overall algorithm. The parameters provided are:

<code>nummyrows</code>	Number of local rows of a nullspace vector on a process
<code>numpde</code>	Maximum number of degrees of freedom per node as specified in the solver options by the user.
<code>dim_nullsp</code>	Number of nullspace vectors expected by the solver. If the user specified the dimension of the problem as ‘3’ on input, the algorithm expects six vectors, if the dimension of the problem was specified as ‘2’ or as ‘1’, the algorithm expects three or one nullspace vector, respectively.

If the method returns `NULL`, *ML*’s default nullspace will be used which might lead to slower convergence rates.

```
virtual const Epetra_Vector* getSolution() = 0;
```

The method returns the initial guess or the latest solution iterate stored by the application. It is used only in cases where the nonlinear multigrid is used as a standalone solver instead of as a preconditioner. If a preconditioner is used, implementing an error message is sufficient.

		rigid body modes::					
		translation in x	translation in y	translation in z	rotation round x	rotation round y	rotation round z
nodal degrees of freedom	x-direction	1	0	0	0	$\mathbf{x}_3 - \hat{\mathbf{x}}_3$	$\hat{\mathbf{x}}_2 - \mathbf{x}_2$
	y-direction	0	1	0	$\hat{\mathbf{x}}_3 - \mathbf{x}_3$	0	$\mathbf{x}_1 - \hat{\mathbf{x}}_1$
	z-direction	0	0	1	$\mathbf{x}_2 - \hat{\mathbf{x}}_2$	$\hat{\mathbf{x}}_1 - \mathbf{x}_1$	0

\mathbf{x} : nodal coordinates
 $\hat{\mathbf{x}}$: coordinates of an arbitrary reference point

Figure 11: Nullspace for continuum discretization

```
virtual bool Get_BlockInfo( int* nblocks, int** blocks,
                           int** block_pde) = 0;
```

The method is used if the ‘VBMETIS’ aggregation scheme is chosen by the user on input, see Section 9.5. This aggregation scheme provides support for variable blocked problems when the number of degrees of freedom per node is non-constant throughout the system of equations. In this case the user has to provide additional information about the nodal block structure of the problem.

`nblocks` (output) Local number of blocks on a process

`blocks` (output) Allocated vector of local length matching the row map obtained by `getMap()`. Contains *global* numbering of blocks where the index base is zero.

`block_pde` (output) Allocated vector of local length matching the row map obtained by `getMap()`. Contains the number of the PDE equation each point row entry belongs to, where the index base is zero.

Both allocated vectors are destroyed by the solver when no longer needed. If the method returns `false` on exit, the aggregation scheme ‘METIS’ is used and a constant block size is assumed. An example for the construction of this block information is given in Fig. 12.

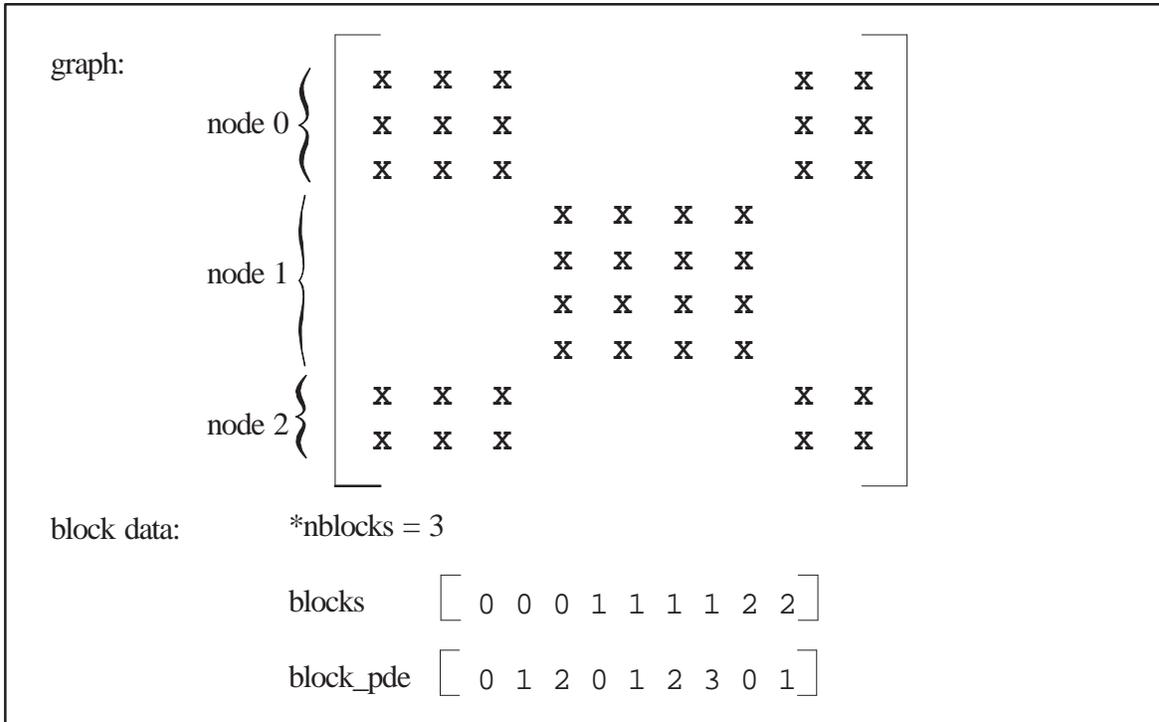


Figure 12: Variable block data

```
bool isNewJacobian() { return isNewJacobian_; }
```

Returns the variable `bool isNewJacobian_` from class `ML_NOX::ML_Nox_Fineinterface`. This variable should be set to `true` by `computeJacobian` whenever a Jacobian is re-evaluated. It should be set to `false` by `computeF` whenever a new residual is evaluated. This allows the solver to determine whether the current Jacobian matches the current residual.

```
int getnumJacobian() { return numJacobian_; }
```

Returns the variable `int numJacobian_` from class `ML_NOX::ML_Nox_Fineinterface`. This variable should be incremented by `computeJacobian` whenever a Jacobian is re-evaluated. It is used for statistical output.

```
void resetsumtime() { t_ = 0.; return; }
```

Used internally to reset the summed time spent in the interface. It is used to measure time spent in the interface and application separately for the solver setup and the iteration phase. It is used for statistical output.

```
int getnumcallscomputeF() { return ncalls_computeF_ }
```

Returns the variable `int ncalls_computeF_` from class `ML_NOX::ML_NoX_Fineinterface`. This variable should be incremented by `computeF` whenever a residual is reevaluated. It is used for statistical output.

```
bool setnumcallscomputeF(int ncalls) { ncalls_computeF_=ncalls;  
                                       return true;}
```

Used internally to reset the variable `int ncalls_computeF_` from class `ML_NOX::ML_NoX_Fineinterface`. It is used to measure the number of calls to `computeF` separately for the setup phase and in the total.

9.3 Nonlinear multigrid as a solver

The nonlinear multigrid class can be used as a solver though in most cases it is recommended to use it as a preconditioner to some outer nonlinear iteration, see Section 9.4. Here we describe the principal setup of the method in Fig. 13 using the solver capability of the class.

The nonlinear multigrid class can be constructed (Fig. 13, line 108) after defining an `Epetra_Comm` derived communicator object, a map reflecting the distribution of solution and residual vectors, rows of the problem graph and the Jacobian. Detailed function documentation is also provided in the `Trilinos/packages/ml/doc` directory of the distribution. Following line 110 in Fig. 13, a variety of options can be chosen which are discussed in detail in Section 9.5.

After all options are chosen and passed to the `ML_NOX::ML_NoX_Preconditioner` class, the `solve()` method starts the setup phase and the iterative solution procedure of the multigrid method.

9.4 Nonlinear multigrid as a preconditioner

It is recommended to use the nonlinear multigrid as a preconditioner to some outer nonlinear Krylov iteration. The basic setup of the preconditioner has already been described in Section 9.3. Additionally, the user has to setup the outer Krylov solver and register the preconditioner with it. As describing the setup of a *NOX* solver would exceed the purpose of this report, we refer to [16] and the example application provided with the distribution and described in Section 9.6.

```

0   #ifdef PARALLEL
1   #include "Epetra_MpiComm.h"
2   #else
3   #include "Epetra_SerialComm.h"
4   #endif
5
6   #include "Epetra_Map.h"
7
8   #include "myinterface.H"
9   #include "ml_nox_preconditioner.H"
(... )
50  #ifdef PARALLEL
51  Epetra_MpiComm  Comm(mpicomm);
52  #else
53  Epetra_SerialComm  Comm();
54  #endif
(... )
100 // create the ML_NOX::ML_Nox_Fineinterface
101 // derived application interface
102 MY_APP_INTERFACE  inter(Comm);
103
104 // get the fine grid row- and vector map
105 Epetra_Map&  map = inter.GetMap();
106
107 // create the nonlinear multigrid class
108 ML_NOX::ML_Nox_Preconditioner  Prec(inter,map,map,Comm);
109
110 // choose options
(... )
150 // solve
151 bool ok = Prec.solve();

```

Figure 13: Setup of the nonlinear algorithm

9.5 Nonlinear preconditioner options

Once the `ML_NOX::ML_Nox_Preconditioner` class is created as shown in Fig. 13, line 108, there exist several methods to pass options to the class and override the default configuration.

In the following, options as well as the methods to pass options to the preconditioner class are listed and discussed. Note that some option choices might conflict or might not have any effect at all depending on how other parameters are chosen. It is therefore recommended to study the variants carefully.

```
bool SetNonlinearMethod( bool islinPrec, int maxlevel,  
                        bool ismatrixfree, bool ismatfreelev0,  
                        bool fixdiagonal, bool constraints);
```

Choice of principal properties of the algorithm.

<code>islinPrec</code>	If <code>true</code> , the preconditioner will act as a standard <i>linear</i> algebraic multigrid preconditioner. If <code>false</code> , the nonlinear FAS scheme (Section 3) will be used. When acting as a linear preconditioner, options for choosing nonlinear smoothers and a nonlinear V-cycle do not take effect.
<code>maxlevel</code>	Maximum number of grids including the given fine grid.
<code>ismatrixfree</code>	If <code>true</code> , the preconditioner will never call the interface to supply a Jacobian matrix. Instead it will use finite differencing to construct a Jacobian itself. Note that finite differencing can be costly and is performed on every grid. Also, as there is no Jacobian present in the setup of the algebraic coarse grid hierarchy, plain aggregation instead of smoothed aggregation multigrid will be used to construct prolongation and restriction operators.
<code>ismatrixfreelev0</code>	If <code>true</code> in combination with <code>ismatrixfree=true</code> , the preconditioner will use finite differencing on the fine grid only and will construct a variational coarse grid hierarchy using smoothed aggregation multigrid. It is recommended to use either <code>ismatrixfree = ismatfreelev0 = true</code> or to use <code>ismatrixfree = ismatfreelev0 = false</code> in case the underlying application can supply a Jacobian matrix.
<code>fixdiagonal</code>	If <code>true</code> , the algorithm will check the Jacobian matrix on the fine grid for rows without any nonzero entries (empty rows). If such an empty row is found, a reasonably sized value will be added to the main diagonal of that row. This might become necessary with some applications though in general no empty rows should appear and the option should therefore be chosen to <code>false</code> by default.
<code>constraints</code>	If <code>true</code> , the algorithm will apply variational constraint enforcement as described in Section 7.

```
bool SetNonlinearSolvers( bool useInCG_fine, bool useInCG,
                        bool useInCG_coarse, bool useBroyden,
                        int niters_fine, int niters,
                        int niters_coarse);
```

Choice of nonlinear smoothing and solving algorithm and the maximum number of iterations to be taken on distinct grids.

```
useInCG_fine,
useInCG,
useInCG_coarse
```

If `true`, the preconditioner will use nonlinear CG as a nonlinear method on the fine, all intermediate and on the coarse grid, respectively. If `false`, quasi-Newton method will be used. Usage of nonlinear CG and Newton's method can be mixed among grids.

```
niters_fine,
niters,
niters_coarse
```

When choosing a quasi-Newton method on a grid, a *linear* CG solve will be performed inside each Newton step. The maximum number of linear CG iterations to be taken can be specified by these variables. These options do not take effect when nonlinear CG is chosen as the nonlinear smoothing method.

```
bool SetPrintLevel(int ml_printlevel);
```

Choice of the amount of output to be generated during setup and iteration. `ml_printlevel` can take values between 0 (no output) and 10 (full output). A value of 6 results in a reasonable amount of information.

```
bool SetDimensions( int spatialDimension, int numPDE, int dimNS);
```

Provide information about the spatial dimension of the problem, the nodal block size and the nullspace to be used.

```
spatialDimension
```

Specify 1, 2 or 3 for 1D, 2D or 3D problems, respectively.

```
numPDE
```

Number of PDE equations. In general equal to the number of degrees of freedom per node on the fine grid and the nodal block size of the Jacobian matrix. If the aggregation scheme "VBMETIS" is used and the problem is of variable block size, the largest block size in the system has to be specified.

```
dimNS
```

Dimension of the approximation to the nullspace of the fine grid nonlinear operator, see Section 2. For 3D structural problems, this usually would be the 6 rigid body modes of the discretized body neglecting Dirichlet boundary conditions. The number of nullspace vectors `dimNS` has to be larger or equal to `numPDE`.

```
bool SetCoarsenType( string coarsenType, int maxlevel,
                    int maxcoarsenSize, int nnodeperagg);
```

Choice of *ML*'s aggregation method in the generation of the coarse grid hierarchy.

coarsenType

Though *ML* supports several other aggregation schemes, the nonlinear preconditioner class currently supports "Uncoupled", "METIS" and "VBMETIS", see also [15]. The "VBMETIS" scheme is currently the only scheme supporting problems with nonconstant nodal blocks, also see Sections 2 and 9.2. "VBMETIS" can also be used for constant block sized problems though the "METIS" scheme is more efficient in this case resulting in an identical grid hierarchy.

maxlevel

Maximum number of levels to be generated. The maximum number of levels to be used depends on the problem size but should be kept as small as possible without resulting in a too large coarse grid problem.

maxcoarsenSize

The setup of the multigrid hierarchy stops generating coarser grids when the current coarsest grid has less than `maxcoarsenSize` equations.

nnodeperagg

When using the "METIS" or "VBMETIS" aggregation scheme, the target number of nodal blocks per aggregate can be specified. Standard choices are 9 in 2D and 27 in 3D. Choosing less nodes per aggregate results in larger and eventually more coarse grids, choosing more nodes per aggregate results in a faster decay of grid size and eventually less coarse grids (a 'cheaper' coarse grid hierarchy) at the cost of reduced convergence rates. The option does not have any effect when the "Uncoupled" aggregation scheme is chosen.

```
bool SetConvergenceCriteria( double FAS_normF, double FAS_nupdate);
```

Choose the convergence criteria norm of the residual vector and norm of the update vector for all grids. The nonlinear smoothing iteration (nonlinear CG or quasi-Newton method) on a grid will terminate successfully when either of these criteria is met. Note that these criteria should be chosen equal or smaller than for the outside nonlinear Krylov method if used as a preconditioner.

```
bool SetRecomputeOffset( int offset );  
bool SetRecomputeOffset( int offset, int recomputestep,  
                          double adaptrecompute, int adaptns);
```

Options to choose when to recompute the multigrid hierarchy. Once the multigrid hierarchy and the Jacobian operators are computed they do not change throughout the nonlinear iteration. In order to speed up the iteration process it might be useful to recompute this data from time to time. Several ways to do so can be chosen using these parameters.

<code>offset</code>	Every <code>offset</code> nonlinear iterations, the multigrid hierarchy and Jacobian operators are destroyed and recomputed from scratch. If no recomputation is desired, <code>offset</code> should be chosen as a very large number.
<code>recomputestep</code>	It might be advantageous to recompute the multigrid hierarchy once after a few nonlinear iterations have taken place. When coming closer to the solution, the approximation quality of the Jacobian operators and the multigrid preconditioner increases. The multigrid hierarchy is recomputed once after the <code>recomputestep</code> iteration. Choosing <code>recomputestep</code> to 0 indicates that the hierarchy is not recomputed.
<code>adaptrecompute</code>	If the initial guess to the nonlinear iteration is far from the solution, the nonlinear multigrid preconditioner might have poor approximation properties and divergence might occur. The multigrid hierarchy is recomputed every time the residual norm of the outside Krylov method is larger than <code>adaptrecompute</code> . Choosing <code>adaptrecompute</code> to 0.0 will not recompute the hierarchy.
<code>adaptns</code>	Number of additional near-nullspace components to be computed by an adaptive smoothed aggregation setup procedure, see also Sections 2, 8.3 and [4]. If chosen to 0, no adaptive setup will be performed.

```
bool SetSmoothers( string finesmoothertype, string smoothertype,  
                  string coarsesolve);
```

Choice of *linear* smoother/preconditioning method to be used as a preconditioner to the *nonlinear* CG or the *linear* CG solve inside Newton method on the fine, intermediate and coarse grid, respectively. Options recognized are "SGS" for domain-decomposition symmetric Gauss-Seidel, "BSGS" for domain decomposition symmetric block Gauss-Seidel, "Jacobi" for damped Jacobi smoothing, "MLS" for polynomial smoothing, "BCheby" for block Chebyshev polynomial smoothing and "AmesosKLU" for a direct solve. Recommended are "MLS" and "SGS", where "AmesosKLU" is supposed to be used on the coarsest grid only. The number of smoothing steps to be taken within one iteration of nonlinear CG or one iteration of *linear* CG inside a Newton step is specified using `SetSmootherSweeps`.

```
bool SetSmootherSweeps( int nsmooth_fine, int nsmooth,  
                       int nsmooth_coarse);
```

Specifies the number of smoothing sweeps of the *linear* smoother specified using `SetSmoothers`. In cases using "AmesosKLU" choosing a 1 is recommended. In case of using "MLS" or "BCheby", the number specified is used as the polynomial order of the smoother. Contrary to linear multigrid methods, it is observed that using more powerful smoothing methods or more smoothing sweeps is efficient in the nonlinear case.

```
bool SetFiniteDifferencing( bool centered, double alpha, double beta);
```

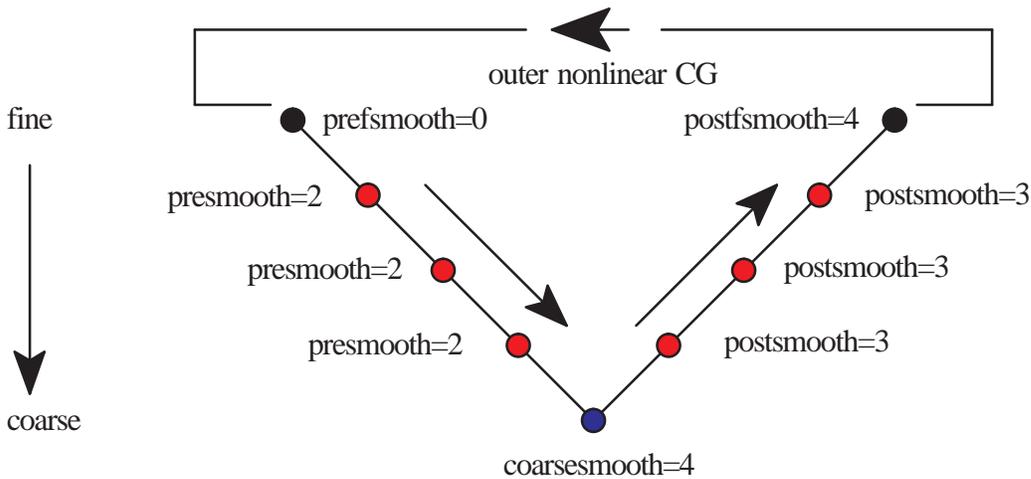
Options to specify parameters of the finite differencing for the Jacobian operators. When `centered=true`, central finite differencing is used at the cost of twice as many residual evaluations (see Section 6). Parameters `alpha` and `beta` are perturbation values from Eq. (26). Finite differencing is always performed using graph coloring. However if the problem has variable nodal block sizes, block collapsed coloring is currently not supported and the algorithm will employ scalar coloring. If it is chosen to use the Jacobian supplied by the underlying application as specified by `SetNonlinearMethod`, no coloring and no finite differencing will be performed.

```
bool SetFAScycle( int pefsmooth, int presmooth,
                 int coarsesmooth, int postsmooth,
                 int postfsmooth, int maxcycle);
```

Options to specify the layout of the potentially nonsymmetric FAS V-cycle. For the finest, intermediate and coarsest grids, the maximum number of presmoothing, coarse grid and postsmoothing iterations can be chosen. Choosing pre- and post iteration numbers differently results in a nonsymmetric V-cycle. Except for `coarsesmooth`, all values can also be independently chosen to be zero, see also Fig. 14a.

When convergence is achieved on some intermediate grid in the presmoothing phase, no coarser grid is visited but instead the algorithm returns to the postsmoothing phase of the next finer grid. For a graphical illustration, see Fig. 14b.

a) nonsymmetric FAS V-cycle, options for pre- and postsmoothing number of iterations



b) convergence during presmoothing step

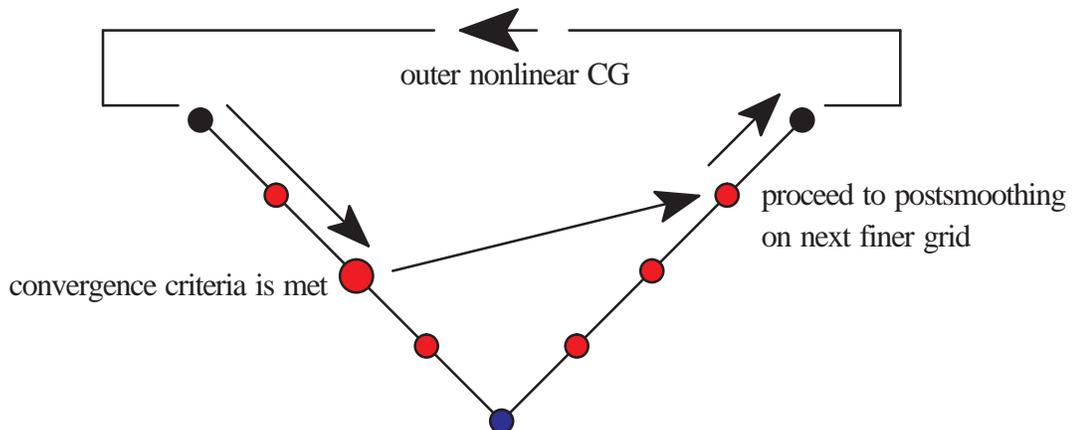


Figure 14: Nonsymmetric FAS v-cycle

9.6 Example application

An example application using the nonlinear multigrid is given in the subdirectory `Trilinos/packages/ml/examples/NonlinML/` of the *Trilinos* installation. It is a simple one dimensional nonlinear finite element problem where the number of elements is specified by the user on the command line. A sufficiently large number (e.g. 10000) must be specified to allow for generation of at least one coarse grid by *ML*.

The file `ml_nox_1DElasticity_example.cpp` contains the main routine where all solver parameters are set. The files `FiniteElementProblem.cpp` and `Problem_Interface.cpp` contain the underlying application and the interface between application and solver, respectively. The latter is a good example on how the interface described in Section 9.2 is implemented while the main routine contains one choice of solver options described in Section 9.5.

10 Conclusion

A nonlinear multigrid solver is described. This description includes algorithm basics as well as detailed user instructions for setting up and directing the solver. More information can be found within *ML*'s documentation and example directories.

11 References

- [1] **Adams, M., Brezina, M., Hu, J., Tuminaro, R. (2003):** *Parallel multigrid smoothing: Polynomial versus Gauss–Seidel*. *J. Comp. Physics*, **188/2**, 593–610.
- [2] **Al–Baali, M., Fletcher, R. (1996):** *On the order of convergence of preconditioned non-linear conjugate gradient methods*. *SIAM J. Sci. Comput.*, **17**, 658–665.
- [3] **Brandt, A. (1977):** *Multi–Level Adaptive Solutions to Boundary Value Problems*. *Math. Comp.*, **31**, 333–390.
- [4] **Brezina, M., Falgout, R., MacLachlan, S., Manteuffel, T., McCormick, S., Ruge, J. (2004):** *Adaptive Smoothed Aggregation (α SA)*. *SIAM J. Sci. Comp.*, **25**, 1896–1920.
- [5] **Büchter, N., Ramm, E., Roehl, D. (1994):** *Three Dimensional Extension of Nonlinear Shell Formulation Based on the Enhanced Assumed Strain Concept*. *Int. J. Num. Meth. Eng.*, **37**, 2551–2568.
- [6] **Briggs, W.L., Henson, V.E., McCormick, S.F. (2000):** *A Multigrid Tutorial, Second Edition*. SIAM Press.

-
- [7] **Fletcher, R. (1987):** *Practical Methods of Optimization, Second Ed.*. Wiley, Chichester, England.
- [8] **Heinstein, M.W., Key, S.W., Blanford, M.L. ():** *A Multigrid Method for Matrix-free Solutions of Non-Linear Quasistatic FE Solid Mechanics Problems*. Draft, Sandia National Laboratories.
- [9] **Heroux, M. Allen, M., Sala, M. (2004):** *An Overview of the Trilinos package*. Technical Report No. SAND2004-1949C, Sandia National Laboratories.
- [10] **Heroux, M. (2005):** *AztecOO User Guide*. Technical Report No. SAND2004-3796, Sandia National Laboratories. software.sandia.gov/trilinos/packages/aztecoo .
- [11] **Heroux, M., Hoekstra, R.J., Williams, A. (2005):** *Epetra User Guide*. Technical Report No. SAND2004-xxxx, Sandia National Laboratories. software.sandia.gov/packages/epetra .
- [12] **Hoekstra, R., Cross, J., Heroux, M., Willenbring, J., Williams, A. (2005):** *EpetraExt linear algebra package*. software.sandia.gov/packages/epetraext .
- [13] **Karypis, G., Kumar, V. (1998):** *METIS 4.0: Unstructured graph partitioning and sparse matrix ordering system*. technical report, Deptment of Computer Science, Univ. of Minnesota.
- [14] **Kelley, C.T. (2003):** *Solving Nonlinear Equations with Newton's Method*. in 'Fundamentals of Algorithms' series, SIAM Press.
- [15] **Sala, M., Tuminaro, R.S., Hu, J.J., Gee, M.W. (2005):** *ML 4.0 Smoothed Aggregation User's Guide*. Technical Report No. SAND2004-4819, Sandia National Laboratories. software.sandia.gov/trilinos/packages/ml .
- [16] **Kolda, T., Pawolwski, R., Bader, B., Hooper, R., Phipps, E., Salinger, A. (2005):** *NOX/LOCA nonlinear solvers and path following algorithms package within Trilinos*. software.sandia.gov/nox .
- [17] **Shewchuk, J.R. (1994):** *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Technical Report, Carnegie Mellon Univ. .
- [18] **Vanek, P., Mandel, J., Brezina, M. (1996):** Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems, *Computing*, **56**, 179-196.
- [19] **Vanek, P., Brezina, M., Tezaur, R. (1999):** Two-Grid Method for Linear Elasticity on Unstructured Meshes, *SIAM Journal on Scientific Computing*, **21**, 900-923.

- [20] **Vanek, P., Brezina, M., Mandel, J. (2001):** Convergence of algebraic multigrid based on smoothed aggregation, *Numerische Mathematik*, **88**, 559–579.
- [21] **Wohlmuth, B.I. (2001):** *Discretization Methods and Iterative Solvers Based on Domain Decomposition*. Lecture Notes in Computational Science and Engineering 17, Springer Press, Berlin, Germany.