

# ElmerSolver

*Setup, execution*

Thomas Zwinger

`thomas.zwinger[at]csc.fi`

Computational Environment & Application

CSC—Scientific Computing Ltd.

The Finnish IT center for science

Espoo, Finland



# Contents

On Bodies, Elements  
and Boundaries

Finite Elements

Linear Solver

Preconditioning

Solution Levels

The Solver Input File (SIF)

Header

Simulation

Solver

Body

Equation

Bodyforce

Material

Initial Conditions

Boundary Conditions

Bodies on Boundaries

Executing Elmer

Tables

MATC

User Defined Functions

User Defined Subroutine

Multiple Meshes

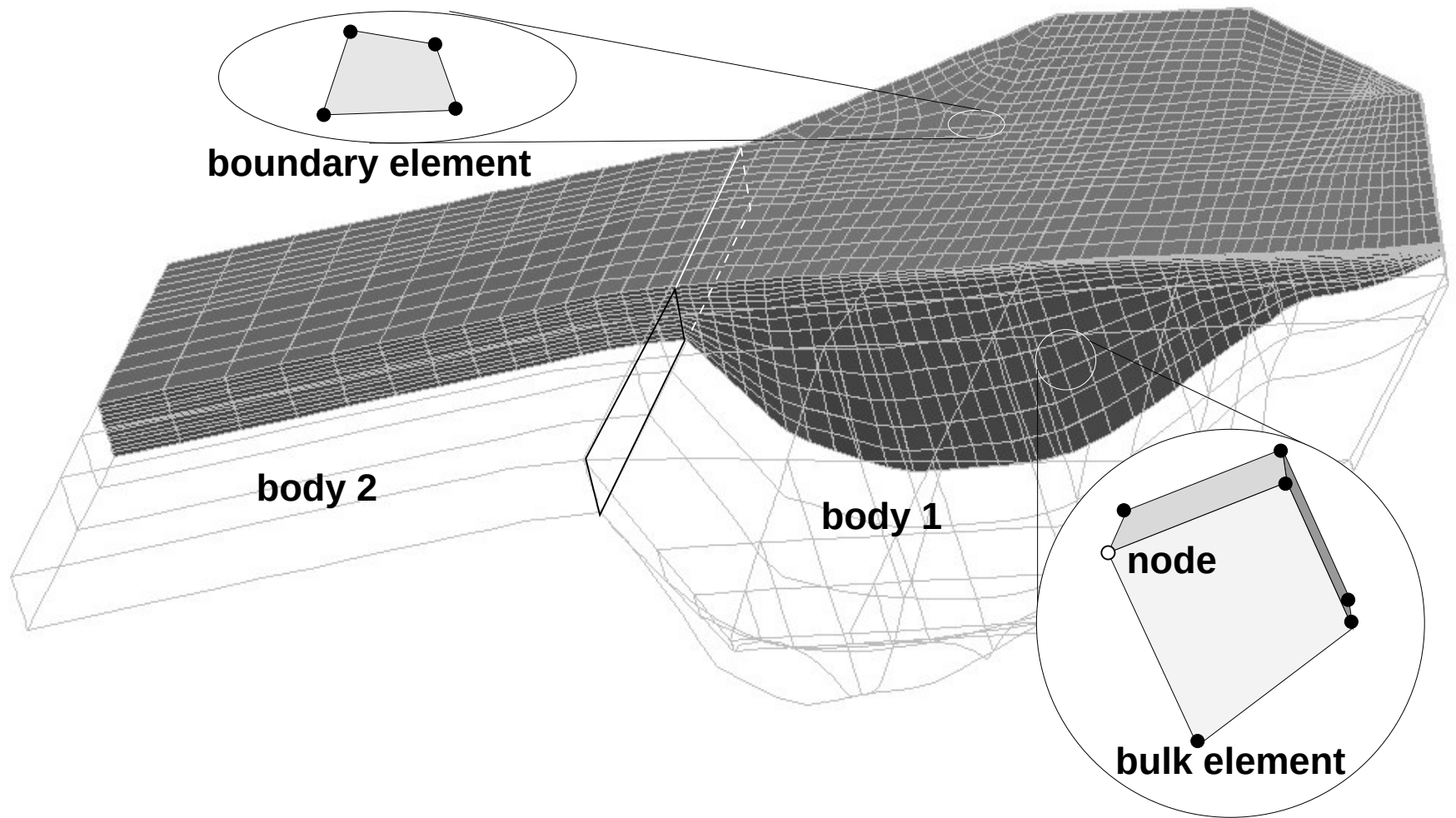
Element Types

Specialities

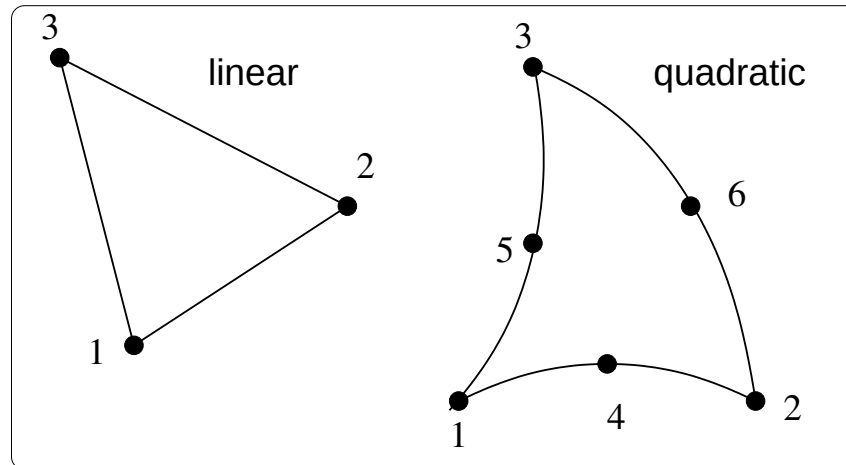
Elmer parallel version



# On Bodies and Boundaries

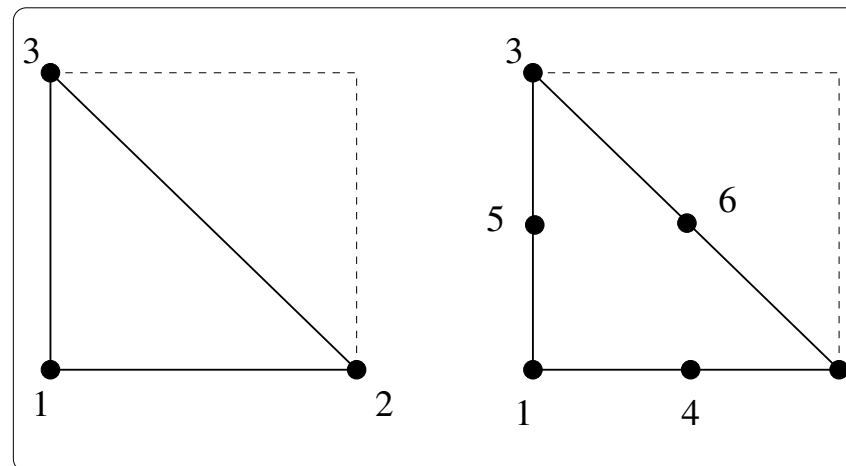


# Finite Elements



Real geometry mesh

+ Coordinate-system metric



Elmer unit size elements

# Finite Elements contd.

General advection-diffusion:

$$\partial\Psi/\partial t + \mathbf{u} \cdot \nabla\Psi = \nabla \cdot (\kappa \nabla\Psi) + \sigma$$

# Finite Elements contd.

General advection-diffusion:

$$\partial\Psi/\partial t + \mathbf{u} \cdot \nabla\Psi = \nabla \cdot (\kappa\nabla\Psi) + \sigma$$

Test function,  $\phi_\alpha$  and integration over domain  $\Omega$ :

$$\int_{\Omega} \partial\Psi/\partial t \phi_\alpha d\Omega + \int_{\Omega} \mathbf{u} \cdot \nabla\Psi \phi_\alpha d\Omega = \int_{\Omega} \nabla \cdot (\kappa\nabla\Psi) \phi_\alpha d\Omega + \int_{\Omega} \sigma \phi_\alpha d\Omega$$

# Finite Elements contd.

General advection-diffusion:

$$\partial\Psi/\partial t + \mathbf{u} \cdot \nabla\Psi = \nabla \cdot (\kappa\nabla\Psi) + \sigma$$

Test function,  $\phi_\alpha$  and integration over domain  $\Omega$ :

$$\int_{\Omega} \partial\Psi/\partial t \phi_\alpha d\Omega + \int_{\Omega} \mathbf{u} \cdot \nabla\Psi \phi_\alpha d\Omega = \int_{\Omega} \nabla \cdot (\kappa\nabla\Psi) \phi_\alpha d\Omega + \int_{\Omega} \sigma \phi_\alpha d\Omega$$

Partial integration of diffusion term:

$$\begin{aligned} \int_{\Omega} \partial\Psi/\partial t \phi_\alpha d\Omega + \int_{\Omega} \mathbf{u} \cdot \nabla\Psi \phi_\alpha d\Omega + \int_{\Omega} \kappa \nabla\Psi \cdot \nabla\phi_\alpha d\Omega = \\ \oint_{\partial\Omega} (\kappa \nabla\Psi \phi_\alpha) \cdot \mathbf{n} d\Omega + \int_{\Omega} \sigma \phi_\alpha d\Omega \end{aligned}$$

# Finite Elements contd.

Discretization of variable::  $\Psi \rightarrow \phi_\beta \Psi_\beta$

# Finite Elements contd.

Discretization of variable::  $\Psi \rightarrow \phi_\beta \Psi_\beta$

and time derivative:  $\partial / \partial t \rightarrow a(\Delta t) \Psi_\beta - b(\Delta t) \Psi_\beta^{t-\Delta t}$

# Finite Elements contd.

Discretization of variable::  $\Psi \rightarrow \phi_\beta \Psi_\beta$

and time derivative:  $\partial / \partial t \rightarrow a(\Delta t) \Psi_\beta - b(\Delta t) \Psi_\beta^{t-\Delta t}$

$$\begin{aligned}
 & \underbrace{\Psi_\beta a(\Delta t) \int_{\Omega} \phi_\beta \phi_\alpha d\Omega}_{\text{M}} + \underbrace{\Psi_\beta \int_{\Omega} [\mathbf{u} \cdot \nabla \phi_\beta \phi_\alpha + \kappa \nabla \phi_\beta \cdot \nabla \phi_\alpha] d\Omega}_{\text{S}} = \\
 & \underbrace{\oint_{\partial\Omega} (\kappa \nabla \Psi \phi_\alpha) \cdot \mathbf{n} d\Omega}_{\text{nat. BC}} + \underbrace{b(\Delta t) \Psi_\beta^{t-\Delta t} \int_{\Omega} \phi_\beta \phi_\alpha d\Omega}_{\text{timeforce}} + \underbrace{\int_{\Omega} \sigma \phi_\alpha d\Omega}_{\text{f}}
 \end{aligned}$$

# Finite Elements contd.

Discretization of variable::  $\Psi \rightarrow \phi_\beta \Psi_\beta$

and time derivative:  $\partial / \partial t \rightarrow a(\Delta t) \Psi_\beta - b(\Delta t) \Psi_\beta^{t-\Delta t}$

$$\underbrace{\Psi_\beta a(\Delta t) \int_{\Omega} \phi_\beta \phi_\alpha d\Omega}_{\mathbf{M}} + \underbrace{\Psi_\beta \int_{\Omega} [\mathbf{u} \cdot \nabla \phi_\beta \phi_\alpha + \kappa \nabla \phi_\beta \cdot \nabla \phi_\alpha] d\Omega}_{\mathbf{S}} =$$

$$\underbrace{\oint_{\partial\Omega} (\kappa \nabla \Psi \phi_\alpha) \cdot \mathbf{n} d\Omega}_{\text{nat. BC}} + \underbrace{b(\Delta t) \Psi_\beta^{t-\Delta t} \int_{\Omega} \phi_\beta \phi_\alpha d\Omega}_{\text{timeforce}} + \underbrace{\int_{\Omega} \sigma \phi_\alpha d\Omega}_{\mathbf{f}}$$

$$(\mathbf{M} + \mathbf{S}) \cdot \Psi = \mathbf{f}$$

$\mathbf{M}$ ... Mass matrix,  $\mathbf{S}$ ... Stiffness matrix,  $\mathbf{f}$ ... force vector

# Linear Solver

- Three solution methods for  $\mathbf{A} \cdot \Psi = \mathbf{f}$
- Linear System Solver = *Keyword*

# Linear Solver

- Three solution methods for  $\mathbf{A} \cdot \Psi = \mathbf{f}$
- Linear System Solver = *Keyword*
- Direct methods (Keyword: **Direct**)
  - Linear System Direct Method =
    - standard LAPACK ( *banded* )
    - alternatively UMFPACK - Unsymmetrical Multi Frontal ( *UMFPACK* )

# Linear Solver

- Three solution methods for  $\mathbf{A} \cdot \Psi = \mathbf{f}$
- Linear System Solver = *Keyword*
- Direct methods (Keyword: **D**irect)
  - Linear System Direct Method =
    - standard LAPACK ( **b**anded )
    - alternatively UMFPACK - Unsymmetrical Multi Frontal ( **U**MFPACK )
- Krylov subspace iterative methods (Keyword: **I**terative)
  - Linear System Iterative Method =  
Conjugate Gradient ( **C**G ), Conjugate Gradient Squared ( **C**GS ), BiConjugate Gradient Stabilized ( **B**iCGStab ), Transpose-Free Quasi-Minimal Residual ( **T**FQMR ), Generalized Minimal Residual ( **G**MRES )

# Linear Solver

- Three solution methods for  $\mathbf{A} \cdot \Psi = \mathbf{f}$
- Linear System Solver = *Keyword*
- Direct methods (Keyword: **D**irect)
  - Linear System Direct Method =
    - standard LAPACK ( **b**anded )
    - alternatively UMFPACK - Unsymmetrical Multi Frontal ( **U**MFPACK )
- Krylov subspace iterative methods (Keyword: **I**terative)
  - Linear System Iterative Method =  
Conjugate Gradient ( **C**G ), Conjugate Gradient Squared ( **C**GS ), BiConjugate Gradient Stabilized ( **B**iCGStab ), Transpose-Free Quasi-Minimal Residual ( **T**FQMR ), Generalized Minimal Residual ( **G**MRES )
- Multilevel (Keyword: **M**ultigrid) Geometric (GMG) and Algebraic (AMG) Multigrid

# Precondition strategies

$$\mathbf{A} \cdot \Psi = \mathbf{f}$$

# Precondition strategies

$$\mathbf{A} \cdot \Psi = \mathbf{f}$$

solving instead:

$$\mathbf{A}\mathbf{M}^{-1} \cdot \Phi = \mathbf{f}, \text{ with } \Phi = \mathbf{M} \cdot \Psi$$

# Precondition strategies

$$\mathbf{A} \cdot \Psi = \mathbf{f}$$

solving instead:

$$\mathbf{A}\mathbf{M}^{-1} \cdot \Phi = \mathbf{f}, \text{ with } \Phi = \mathbf{M} \cdot \Psi$$

Why?:  $\mathbf{A}\mathbf{M}^{-1}$  shall have a towards convergence of iterative methods improved spectrum

# Precondition strategies

$$\mathbf{A} \cdot \boldsymbol{\Psi} = \mathbf{f}$$

solving instead:

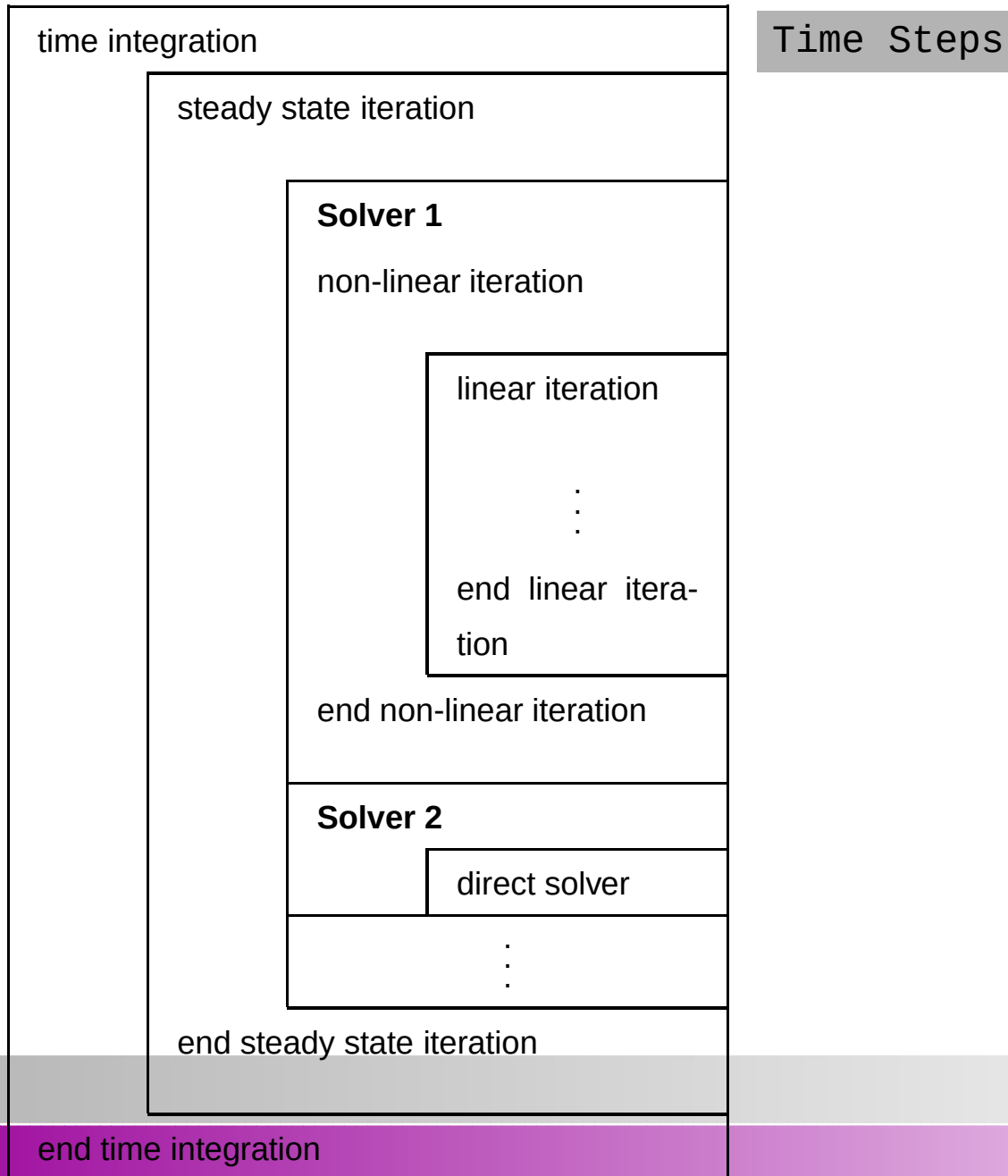
$$\mathbf{A}\mathbf{M}^{-1} \cdot \boldsymbol{\Phi} = \mathbf{f}, \text{ with } \boldsymbol{\Phi} = \mathbf{M} \cdot \boldsymbol{\Psi}$$

Why?:  $\mathbf{A}\mathbf{M}^{-1}$  shall have a towards convergence of iterative methods improved spectrum

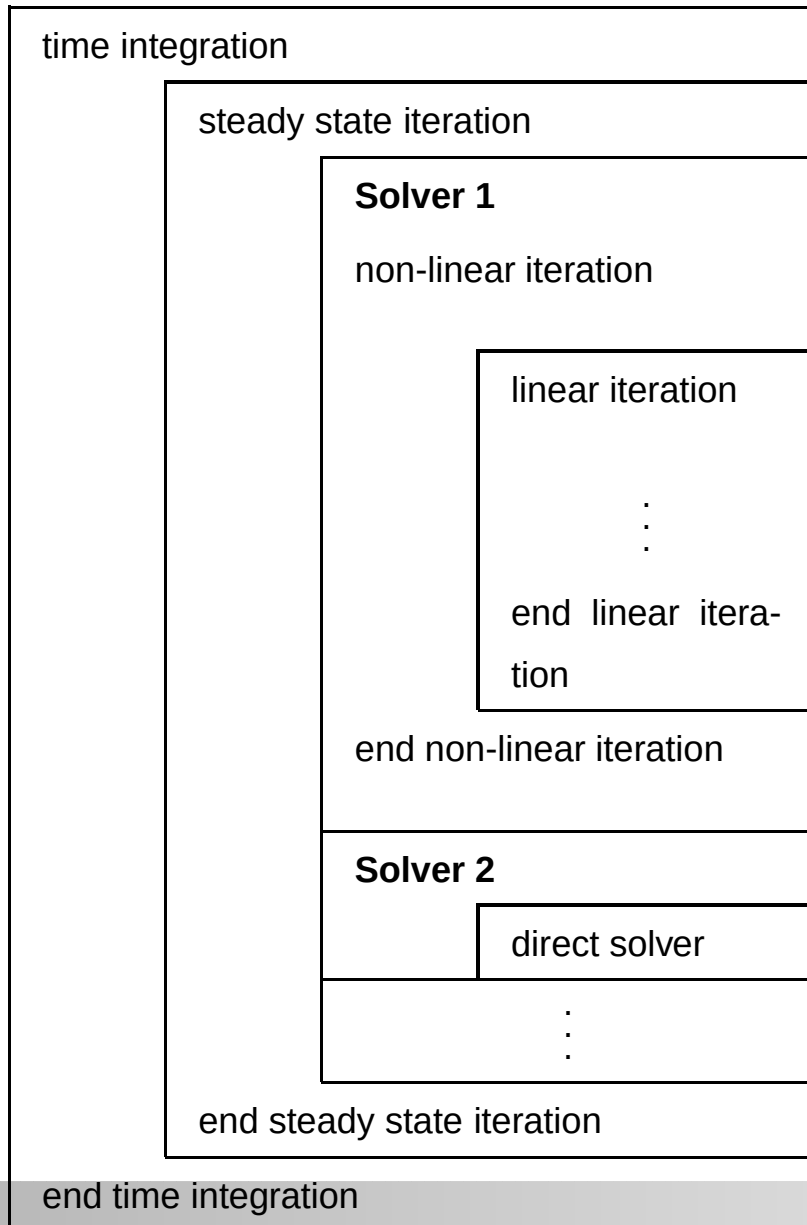
Linear System Preconditioning =

- None
- Diagonal
- ILU $n$   $n = 0, 1, 2, \dots$
- ILUT
- Multigrid

# Solution Levels



# Solution Levels



Time Steps

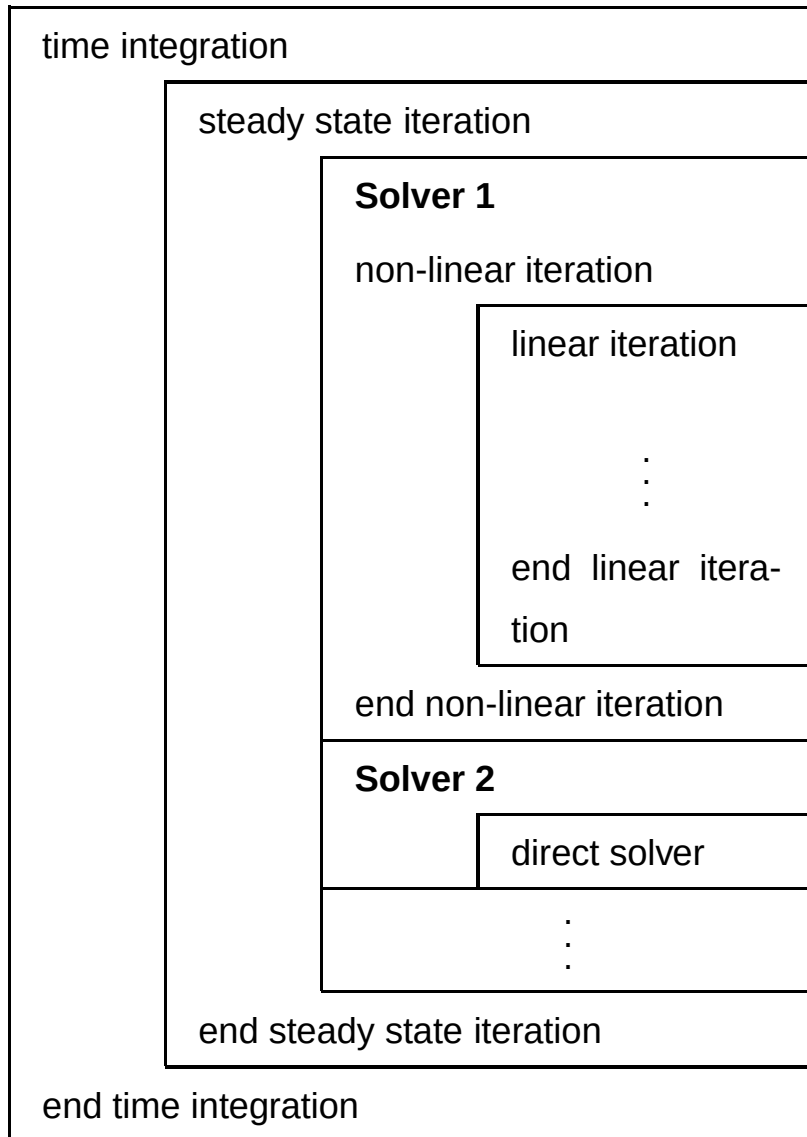
Steady State Max Iterations

Steady State Convergence Tolerance



CSC

# Solution Levels



Time Steps

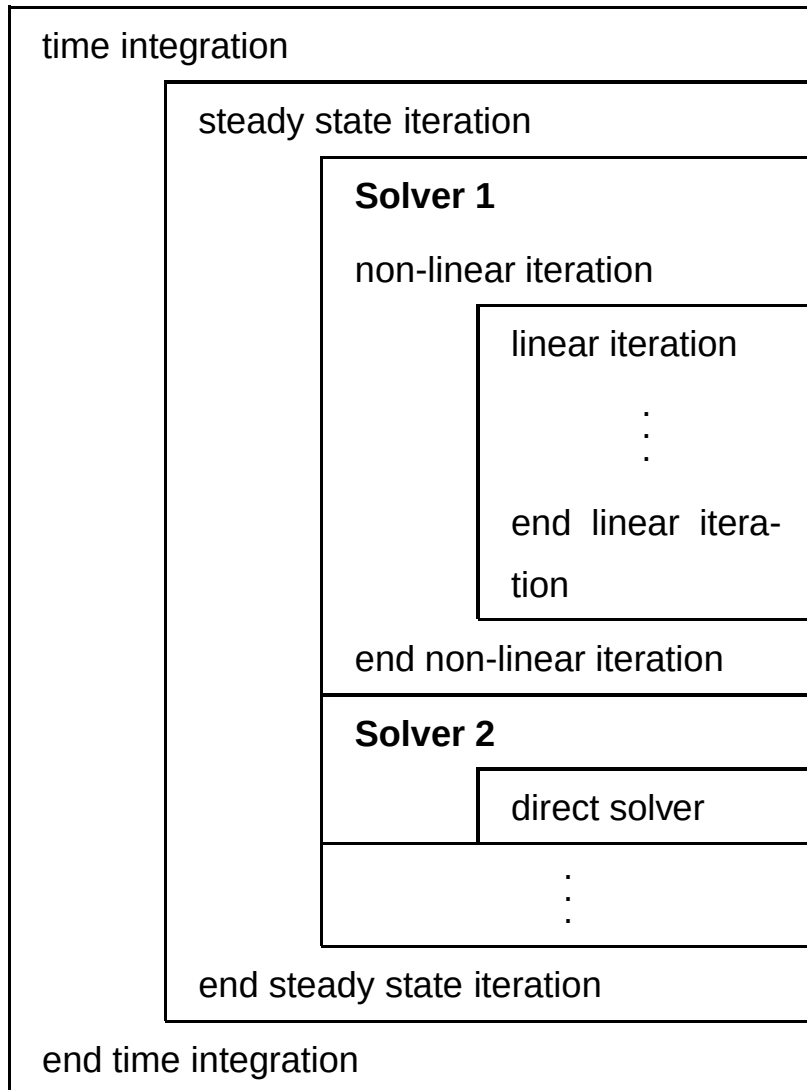
Steady State Max Iterations

Nonlinear Max Iterations

Nonlinear System Convergence Tolerance

Steady State Convergence Tolerance

# Solution Levels



Time Steps

Steady State Max Iterations

Nonlinear Max Iterations

Linear System Max Iterations

Linear System Convergence Tolerance

Nonlinear System Convergence Tolerance

Steady State Convergence Tolerance

# The Solver Input File (SIF)

- contains all the information for the solution step,  
`ElmerSolver_mpi`

# The Solver Input File (SIF)

- contains all the information for the solution step,  
`ElmerSolver_mpi`
- can be exported by `ElmerFront` ...

# The Solver Input File (SIF)

- contains all the information for the solution step,  
`ElmerSolver_mpi`
  - can be exported by `ElmerFront` ...
- ... but simply also composed using a text editor

# The Solver Input File (SIF)

- contains all the information for the solution step,  
`ElmerSolver_mpi`
  - can be exported by `ElmerFront` ...
- ... but simply also composed using a text editor

## The Rules:

# The Solver Input File (SIF)

- contains all the information for the solution step, `ElmerSolver_mpi`
- can be exported by `ElmerFront` ...
- ... but simply also composed using a text editor

## The Rules:

- comments start with `!`

# The Solver Input File (SIF)

- contains all the information for the solution step, `ElmerSolver_mpi`
- can be exported by `ElmerFront` ...
- ... but simply also composed using a text editor

## The Rules:

- comments start with `!`
- Important: do not use tabulators for indents!

# The Solver Input File (SIF)

- contains all the information for the solution step, `ElmerSolver_mpi`
- can be exported by `ElmerFront` ...
- ... but simply also composed using a text editor

## The Rules:

- comments start with `!`
- Important: do not use tabulators for indents!
- a section always ends with the keyword `End`

# The Solver Input File (SIF)

- contains all the information for the solution step, `ElmerSolver_mpi`
- can be exported by `ElmerFront` ...
- ... but simply also composed using a text editor

## The Rules:

- comments start with `!`
- Important: do not use tabulators for indents!
- a section always ends with the keyword `End`
- parameters (except from Elmer keyword database) need to be casted by their types: `Integer` `Real` `Logical` `String` `File`

# The Solver Input File (SIF)

- contains all the information for the solution step, `ElmerSolver_mpi`
- can be exported by `ElmerFront` ...
- ... but simply also composed using a text editor

## The Rules:

- comments start with `!`
- Important: do not use tabulators for indents!
- a section always ends with the keyword `End`
- parameters (except from Elmer keyword database) need to be casted by their types: `Integer` `Real` `Logical` `String` `File`
- `Parametername(n, m)` indicates a  $n \times m$  array

# Header

The header declares where to search for the mesh database

Header

Mesh DB "." "dirname"

End

preceding path + directory name of mesh database

# Constants

Declaration of constant values that can be obtained from within **every** solver and boundary condition **subroutine** or **function**, can be declared.

```
Constants
```

```
Gas Constant = Real 8.314E00
```

```
Gravity (4) = 0 -1 0 9.81
```

```
End
```

a scalar constant

Gravity vector, an array with a registered name

# Simulation

## Principle declarations for simulation

Simulation

Coordinate System = "Cartesian 2D"

Coordinate Mapping(3) = Integer 1 2 3

Simulation Type = "Steady"

Output Intervals = 1

Steady State Max Iterations = 10

Steady State Min Iterations = 2

Output File = "name.result"

Post File = "name.ep"

max output level =  $n$

End

choices: Cartesian {1D,2D,3D},  
Polar {2D,3D}, Cylindric,  
Cylindric Symmetric, Axi  
Symmetric

permute, if you want to interchange directions

either Steady or Transient

how often you want to have results

maximum rounds on one time level

minimum rounds on one Timestep

contains data to restart run

ElmerPost-file

$n=1$  talkative like a Finnish lumberjack,

$n=42$  all and everything

# Solver

## Example: (Navier) Stokes solver

Solver 1

Equation = "Navier-Stokes"

Linear System Solver = "Direct"

Linear System Direct Method = "UMFPack"

Linear System Convergence Tolerance = 1.0E-06

Linear System Abort Not Converged = True

Linear System Preconditioning = "ILU2"

Steady State Convergence Tolerance = 1.0E-03

Stabilization Method = Stabilized

Nonlinear System Convergence Tolerance = 1.0E-05

Nonlinear System Max Iterations = 1

Nonlinear System Min Iterations = 1

Nonlinear System Newton After Iterations = 30

Nonlinear System Newton After Tolerance = 1.0E-05

End

name of the solver

alt. Iterative

a linear problem



# Body

Here the different bodies (there can be more than one) get their Equation, Material, Body Force and Initial Condition assigned

```
Body 1
```

```
  Name = "identifier"
```

give the body a name

```
  Equation = 1
```

```
  Material = 1
```

```
  Body Force = 1
```

```
  Initial Condition = 1
```

```
End
```

# Equation

- set active solvers
- give keywords for the behaviour of different solvers

```
Equation 1
```

```
Active Solvers(2) = 1 2
```

```
Convection = Computed
```

```
Flow Solution Name = String "Flow Solution"
```

```
NS Convect = False
```

```
End
```

# Bodyforce

- declares the solver-specific  $\mathbf{f}$  from  $\mathbf{A} \cdot \Psi = \mathbf{f}$  for the body
- body force can also be a dependent function (see later).

Here for the (Navier) Stokes solver

```
Body Force 1
  Flow BodyForce 1 = 0.0
  Flow BodyForce 2 = -9.81 ! good old gravity
End
```

# Material

- sets material properties for the body.
- material properties can be scalars or tensors and also
- can be given as dependent functions

```
Material 1
```

```
    Viscosity = 1.0E13
```

```
    Density = 918.0
```

```
    My Heat Capacity = Real 1002.0
```

```
End
```

not in keyword DB!

# Initial Conditions

- initializes variable values
- sets initial guess for steady state simulation
- sets initial value for transient simulation
- variable values can be functions

```
Initial Condition 1
```

```
Velocity 1 = 0.0
```

```
Velocity 2 = 1.0
```

```
Pressure = 0.0
```

```
My Variable = Real 0.0
```

not in keyword DB

```
End
```

# Boundary Conditions

- Dirichlet: `variablename = value`
- Neumann: often enabled with keyword (e.g., HTEqu. Heat Flux BC = True) followed by the flux value
- No BC  $\equiv$  Natural BC!
- values can be given as functions

Example: (Navier) Stokes with no penetration (normal) and free slip (tangential)

```
Boundary Condition 1
```

```
    Name = "slip"
```

```
    Target Boundaries = 4
```

```
    Normal-Tangential Velocity = Logical True
```

```
    Velocity 1 = Real 0.0
```

```
End
```

name

refers to boundary no. 4 in mesh

components with respect to surface normal

normal component

# Bodies on Boundaries

- need to solve (dimension-1) PDEs (e.g., kinematic BC on free surface)

# Bodies on Boundaries

- need to solve (dimension-1) PDEs (e.g., kinematic BC on free surface)
- need to define the (dimension-1) entity as a separate body

# Bodies on Boundaries

- need to solve (dimension-1) PDEs (e.g., kinematic BC on free surface)
- need to define the (dimension-1) entity as a separate body
- in the corresponding **Boundary**-section:  
`Body ID =  $n$`  with  $n >$  highest occurring body in the mesh

# Bodies on Boundaries

- need to solve (dimension-1) PDEs (e.g., kinematic BC on free surface)
- need to define the (dimension-1) entity as a separate body
- in the corresponding **Boundary**-section:  
`Body ID =  $n$`  with  $n >$  highest occurring body in the mesh
- define Body Force, Material, Equation and Initial Condition for that body

# Bodies on Boundaries

- need to solve (dimension-1) PDEs (e.g., kinematic BC on free surface)
- need to define the (dimension-1) entity as a separate body
- in the corresponding **Boundary**-section:  
`Body ID =  $n$`  with  $n >$  highest occurring body in the mesh
- define Body Force, Material, Equation and Initial Condition for that body
- full dimensional metric is still valid on the BC body  $\Rightarrow$  has to be taken into account in user supplied subroutines

# Executing Elmer

- make sure that the directory `$ELMER_HOME/bin` is in the path of your OS

# Executing Elmer

- make sure that the directory `$ELMER_HOME/bin` is in the path of your OS
- **serial**  
either launch directly, with `ElmerSolver mysolverinputfile.sif`

# Executing Elmer

- make sure that the directory `$ELMER_HOME/bin` is in the path of your OS
- **serial**
  - either launch directly, with `ElmerSolver mysolverinputfile.sif`
  - or write create a file `ELMERSOLVER_STARTINFO` in the directory where you launch from

# Executing Elmer

- make sure that the directory `$ELMER_HOME/bin` is in the path of your OS
- **serial**
  - either launch directly, with `ElmerSolver mysolverinputfile.sif`
  - or write create a file `ELMERSOLVER_STARTINFO` in the directory where you launch from
  - e.g. `echo mysolverinputfile.sif > ELMERSOLVER_STARTINFO`

# Executing Elmer

- make sure that the directory `$ELMER_HOME/bin` is in the path of your OS
- **serial**
  - either launch directly, with `ElmerSolver mysolverinputfile.sif`
  - or write create a file `ELMERSOLVER_STARTINFO` in the directory where you launch from
  - e.g. `echo mysolverinputfile.sif > ELMERSOLVER_STARTINFO`
  - launch simply with `ElmerSolver`

# Executing Elmer

- make sure that the directory `$ELMER_HOME/bin` is in the path of your OS
- **serial**
  - either launch directly, with `ElmerSolver mysolverinputfile.sif`
  - or write create a file `ELMERSOLVER_STARTINFO` in the directory where you launch from
  - e.g. `echo mysolverinputfile.sif > ELMERSOLVER_STARTINFO`
  - launch simply with `ElmerSolver`
- **parallel**
  - needs `ELMERSOLVER_STARTINFO` plus evtl. a host-file (if distributed execution)

# Executing Elmer

- make sure that the directory `$ELMER_HOME/bin` is in the path of your OS
- **serial**
  - either launch directly, with `ElmerSolver mysolverinputfile.sif`
  - or write create a file `ELMERSOLVER_STARTINFO` in the directory where you launch from
  - e.g. `echo mysolverinputfile.sif > ELMERSOLVER_STARTINFO`
  - launch simply with `ElmerSolver`
- **parallel**
  - needs `ELMERSOLVER_STARTINFO` plus evtl. a host-file (if distributed execution)
  - launch with `mpirun -np 4 --hostfile hostfilename ElmerSolver`

# Tables and Arrays

- **Tables** may be used for piecewise linear dependency of a variable

# Tables and Arrays

- **Tables** may be used for piecewise linear dependency of a variable

```
Density = Variable Temperature
Real
    0    900
    273  1000
    300  1020
    400  1000
End
```

# Tables and Arrays

- **Tables** may be used for piecewise linear dependency of a variable

```
Density = Variable Temperature
Real
    0    900
    273  1000
    300  1020
    400  1000
End
```

- **Arrays** may be used to declare vector/tensor parameters

# Tables and Arrays

- **Tables** may be used for piecewise linear dependency of a variable

```
Density = Variable Temperature
Real
    0    900
    273  1000
    300  1020
    400  1000
End
```

- **Arrays** may be used to declare vector/tensor parameters

```
Target Boundaries(3) = 2 4 5
My Parameter Array(3,3) = Real 1 2 3 \
                                4 5 6 \
                                7 8 9
```

# MATC

- library for the numerical evaluation of mathematical expressions

# MATC

- library for the numerical evaluation of mathematical expressions
- defined in SIF for use in ElmerSolver

# MATC

- library for the numerical evaluation of mathematical expressions
- defined in SIF for use in `ElmerSolver`  
or by `ElmerPost` as post-processing feature

e.g.  $K \rightarrow ^\circ C$ : `math Celsius = Temperature + 273.16`

# MATC

- library for the numerical evaluation of mathematical expressions
- defined in SIF for use in **E<sub>l</sub>merSo<sub>l</sub>ver**  
or by **E<sub>l</sub>merPost** as post-processing feature  
e.g.  $K \rightarrow ^\circ C$ : `math Celsius = Temperature + 273.16`
- very close to C-syntax

# MATC

- library for the numerical evaluation of mathematical expressions
- defined in SIF for use in `ElmerSolver`  
or by `ElmerPost` as post-processing feature  
e.g.  $K \rightarrow ^\circ C$ : `math Celsius = Temperature + 273.16`
- very close to C-syntax  
also logical evaluations (if) and loops (for)

# MATC

- library for the numerical evaluation of mathematical expressions
- defined in SIF for use in `ElmerSolver`  
or by `ElmerPost` as post-processing feature  
e.g.  $K \rightarrow ^\circ C$ : `math Celsius = Temperature + 273.16`
- very close to C-syntax  
also logical evaluations (if) and loops (for)
- documentation on Funet ([MATC Manual](#))

# MATC contd.

- simple numerical evaluation:

```
Viscosity Exponent = Real MATC "1.0/3.0" or
```

```
Viscosity Exponent = Real $1.0/3.0
```

# MATC contd.

- simple numerical evaluation:

```
Viscosity Exponent = Real MATC "1.0/3.0" or
```

```
Viscosity Exponent = Real $1.0/3.0
```

- as a function dependent on a variable:

```
Heat Capacity = Variable Temperature
```

```
Real MATC "2.1275D03 + 7.253D00*(tx - 273.16)"
```

# MATC contd.

- simple numerical evaluation:

```
Viscosity Exponent = Real MATC "1.0/3.0" or
```

```
Viscosity Exponent = Real $1.0/3.0
```

- as a function dependent on a variable:

```
Heat Capacity = Variable Temperature
```

```
Real MATC "2.1275D03 + 7.253D00*(tx - 273.16)"
```

- as a function of multiple variables:

```
Temp = Variable Latitude, Coordinate 3
```

```
Real MATC "49.13 + 273.16 - 0.7576 * tx(0) - 7.992E-03 * tx(1)"
```

# MATC contd.

- simple numerical evaluation:

```
Viscosity Exponent = Real MATC "1.0/3.0" or
```

```
Viscosity Exponent = Real $1.0/3.0
```

- as a function dependent on a variable:

```
Heat Capacity = Variable Temperature
```

```
Real MATC "2.1275D03 + 7.253D00*(tx - 273.16)"
```

- as a function of multiple variables:

```
Temp = Variable Latitude, Coordinate 3
```

```
Real MATC "49.13 + 273.16 - 0.7576 * tx(0) - 7.992E-03 * tx(1)"
```

- as function defined before header:

```
$ function stemp(X) { _stemp = 49.13 + 273.16 - 0.7576*X(0)
```

```
- 7.992E-03*X(1) }
```

```
Temp = Variable Latitude, Coordinate 3
```

```
Real MATC "stemp(tx)"
```

# MATC contd.

- simple numerical evaluation:

```
Viscosity Exponent = Real MATC "1.0/3.0" or
```

```
Viscosity Exponent = Real $1.0/3.0
```

- as a function dependent on a variable:

```
Heat Capacity = Variable Temperature
```

```
Real MATC "2.1275D03 + 7.253D00*(tx - 273.16)"
```

- as a function of multiple variables:

```
Temp = Variable Latitude, Coordinate 3
```

```
Real MATC "49.13 + 273.16 - 0.7576 * tx(0) - 7.992E-03 * tx(1)"
```

- as function defined before header:

```
$ function stemp(X) { _stemp = 49.13 + 273.16 - 0.7576*X(0)
```

```
- 7.992E-03*X(1) }
```

```
Temp = Variable Latitude, Coordinate 3
```

```
Real MATC "stemp(tx)"
```

# User Defined Functions

Example:  $\rho(T(^{\circ}C)) = 1000 \cdot [1 - 10^{-4} \cdot (T - 273.0)]$

# User Defined Functions

Example:  $\rho(T(^{\circ}C)) = 1000 \cdot [1 - 10^{-4} \cdot (T - 273.0)]$

```
FUNCTION getdensity( Model, n, T ) RESULT(dens)
USE DefUtils
IMPLICIT None
  TYPE(Model_t) :: Model

  INTEGER :: n

  REAL(KIND=dp) :: T, dens

  dens = 1000*(1-1.0d-4(T-273.0d0))

END FUNCTION getdensity
```

# User Defined Functions

Example:  $\rho(T(^{\circ}C)) = 1000 \cdot [1 - 10^{-4} \cdot (T - 273.0)]$

```
FUNCTION getdensity( Model, n, T ) RESULT(dens)
USE DefUtils
IMPLICIT None
  TYPE(Model_t) :: Model

  INTEGER :: n

  REAL(KIND=dp) :: T, dens

  dens = 1000*(1-1.0d-4(T-273.0d0))

END FUNCTION getdensity
```

compile: `elmerf90 mydensity.f90 -o mydensity`

# User Defined Functions

Example:  $\rho(T(^{\circ}C)) = 1000 \cdot [1 - 10^{-4} \cdot (T - 273.0)]$

```
FUNCTION getdensity( Model, n, T ) RESULT(dens)
USE DefUtils
IMPLICIT None
  TYPE(Model_t) :: Model

  INTEGER :: n

  REAL(KIND=dp) :: T, dens

  dens = 1000*(1-1.0d-4(T-273.0d0))

END FUNCTION getdensity
```

compile: `elmerf90 mydensity.f90 -o mydensity`

SIF: Density = Variable Temperature  
Procedure "mydensity" "getdensity"

# User Defined Subroutines

```
RECURSIVE SUBROUTINE &  
mysolver( Model,Solver,dt,TransientSimulation )  
TYPE(Model_t) :: Model  
TYPE(Solver_t) :: Solver  
REAL(KIND=dp) :: dt  
LOGICAL :: TransientSimulation  
...  
assembly, solution  
...  
END SUBROUTINE mysolver
```

# User Defined Subroutines

```
RECURSIVE SUBROUTINE &  
mysolver( Model,Solver,dt,TransientSimulation )  
TYPE(Model_t) :: Model  
TYPE(Solver_t) :: Solver  
REAL(KIND=dp) :: dt  
LOGICAL :: TransientSimulation  
...  
assembly, solution  
...  
END SUBROUTINE mysolver
```

Model      pointer to the whole Model (solvers, variables)

Solver      pointer to the particular solver

dt          current time step size

TransientSimulation      .TRUE. if transient simulation

# User Defined Subroutines

```
RECURSIVE SUBROUTINE &  
mysolver( Model,Solver,dt,TransientSimulation )  
TYPE(Model_t) :: Model  
TYPE(Solver_t) :: Solver  
REAL(KIND=dp) :: dt  
LOGICAL :: TransientSimulation  
...  
assembly, solution  
...  
END SUBROUTINE mysolver
```

Model      pointer to the whole Model (solvers, variables)

Solver      pointer to the particular solver

dt          current time step size

TransientSimulation    .TRUE. if transient simulation

compile:      `elmerf90 mysolverfile.f90 -o mysolverexe`



# User Defined Subroutines

```
RECURSIVE SUBROUTINE &  
mysolver( Model,Solver,dt,TransientSimulation )  
TYPE(Model_t) :: Model  
TYPE(Solver_t) :: Solver  
REAL(KIND=dp) :: dt  
LOGICAL :: TransientSimulation  
...  
assembly, solution  
...  
END SUBROUTINE mysolver
```

Model      pointer to the whole Model (solvers, variables)

Solver      pointer to the particular solver

dt          current time step size

TransientSimulation    .TRUE. if transient simulation

compile:

```
elmerf90 mysolverfile.f90 -o mysolverexe
```

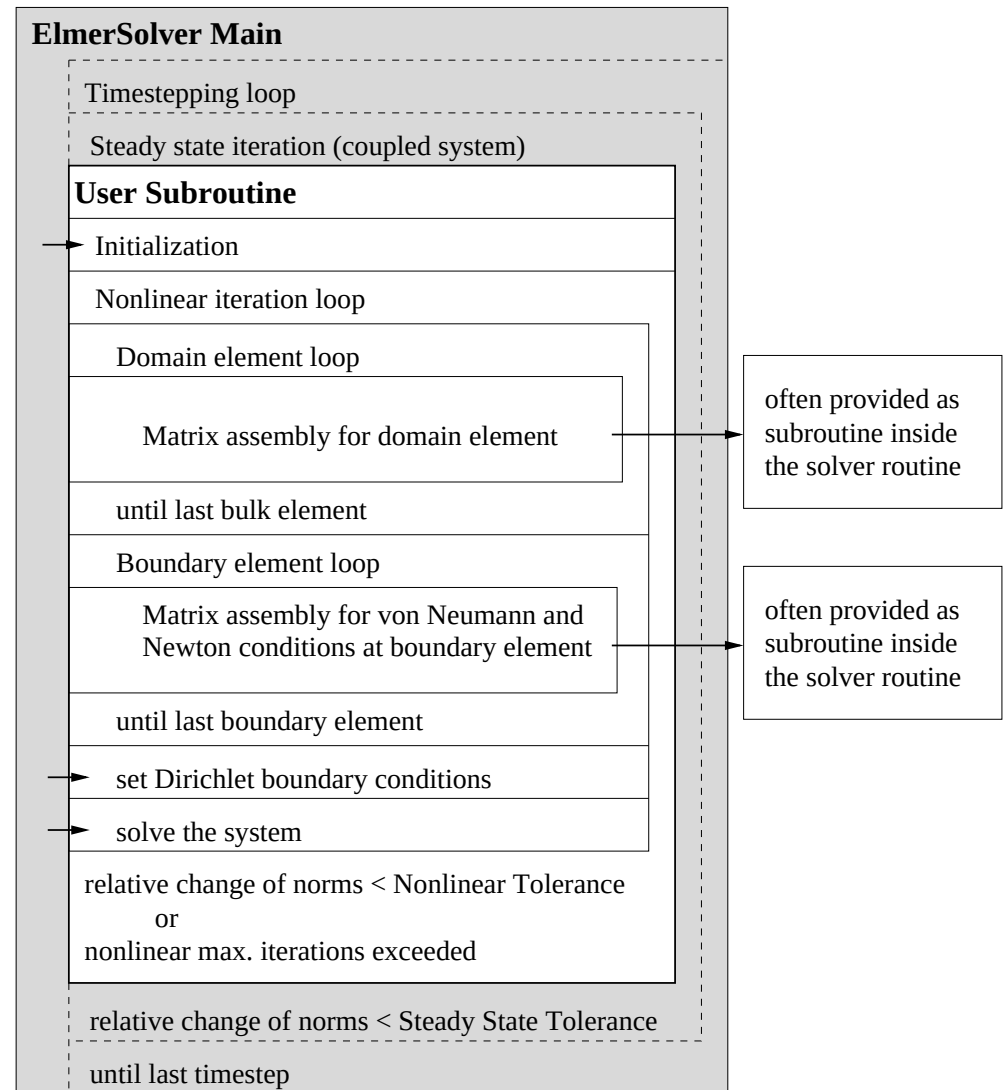
```
Procedure = "/path/to/mysolverexe" "mysolver"
```

SIF:



CSC

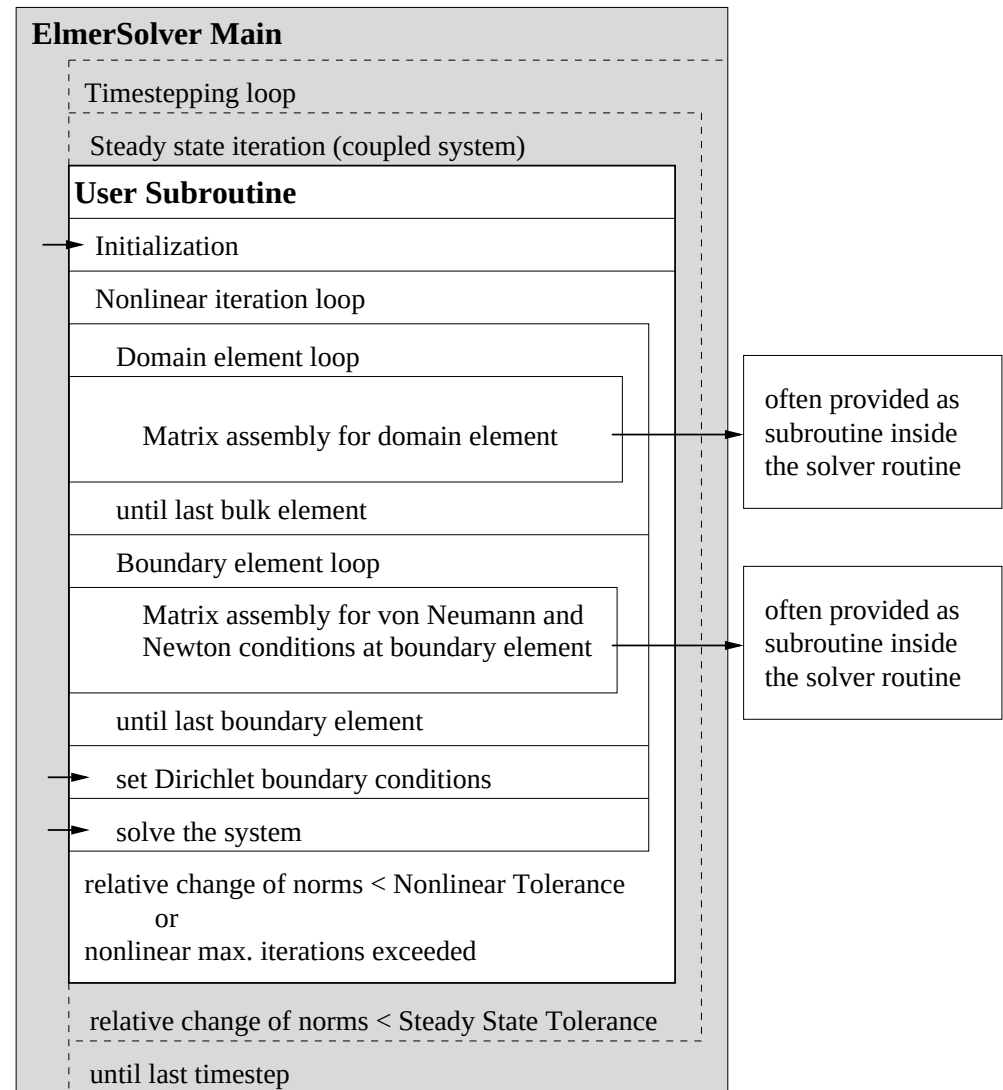
# User Defined Subroutines contd.



# User Defined Subroutines contd.

## Pre-defined routines

- CALL  
DefaultInitialize()
- CALL  
DefaultUpdateEquations(  
STIFF, FORCE )
- CALL  
DefaultFinishAssembly()
- CALL  
DefaultDirichletBCs()
- Norm =  
DefaultSolve()



# Multiple Meshes

- In the **Header**, declare the *global* mesh database

```
Mesh DB "." "dirname"
```

# Multiple Meshes

- In the **Header**, declare the *global* mesh database

```
Mesh DB "." "dirname"
```

- In the **Solver**, declare the *local* mesh the solver is run on:

```
Mesh = File "/path/to/" "mesh"
```

# Multiple Meshes

- In the **Header**, declare the *global* mesh database

```
Mesh DB "." "dirname"
```

- In the **Solver**, declare the *local* mesh the solver is run on:

```
Mesh = File "/path/to/" "mesh"
```

- variable values will be interpolated

# Multiple Meshes

- In the **Header**, declare the *global* mesh database

```
Mesh DB "." "dirname"
```

- In the **Solver**, declare the *local* mesh the solver is run on:

```
Mesh = File "/path/to/" "mesh"
```

- variable values will be interpolated



they will boldly be extrapolated, should your meshes not be congruent!

# Element Types

- In section Equation:

# Element Types

- In section Equation:

```
Element = [n:#dofs d:#dofs p:#dofs b:#dofs e:#dofs f:#dofs]
```

n ... nodal, d ... DG element, p p-element, b ... bubble, e ... edge, f ... face DOFs

# Element Types

- In section Equation:

```
Element = [n:#dofs d:#dofs p:#dofs b:#dofs e:#dofs f:#dofs]
```

n ... nodal, d ... DG element, p p-element, b ... bubble, e ... edge, f ... face DOFs

- ```
Element = [d:0]
```

 ... DG DOFs  $\equiv$  mesh element nodes

# Element Types

- In section Equation:

```
Element = [n:#dofs d:#dofs p:#dofs b:#dofs e:#dofs f:#dofs]
```

n ... nodal, d ... DG element, p p-element, b ... bubble, e ... edge, f ... face DOFs

- ```
Element = [d:0]
```

 ... DG DOFs  $\equiv$  mesh element nodes

- If Equation applies to more than one solver, 

```
Element = ...
```

 applies for all solver.



# Element Types

- In section Equation:

```
Element = [n:#dofs d:#dofs p:#dofs b:#dofs e:#dofs f:#dofs]
```

n ... nodal, d ... DG element, p p-element, b ... bubble, e ... edge, f ... face DOFs

- ```
Element = [d:0]
```

 ... DG DOFs  $\equiv$  mesh element nodes

- If Equation applies to more than one solver, 

```
Element = ...
```

 applies for all solver.

selectively for each solver:

```
Element[1] = ...  
Element[2] = ...  
:  
:  
Element[n] = ...
```

# Specialities

- given names for components of vector fields:

```
Variable = var_name[cname 1:#dofs cname 2:#dofs ... ]
```

# Specialities

- given names for components of vector fields:

```
Variable = var_name[cname 1:#dofs cname 2:#dofs ... ]
```

- "internal" Solver can be run as external Procedures (enabling definition of variable names)

```
Procedure = "FlowSolve" "FlowSolver"
```

```
Variable = Flow[Veloc:3 Pres:1]
```

# Specialities

- given names for components of vector fields:

```
Variable = var_name[cname 1:#dofs cname 2:#dofs ... ]
```

- "internal" Solver can be run as external Procedures (enabling definition of variable names)

```
Procedure = "FlowSolve" "FlowSolver"
```

```
Variable = Flow[Veloc:3 Pres:1]
```

- Residuals of solver variables (e.g., Navier Stokes):

```
Procedure = "FlowSolve" "FlowSolver"
```

```
Variable = Flow[Veloc:3 Pres:1]
```

```
Exported Variable 1 = Flow Loads[Stress Vector:3 CEQ Residual:1]
```

# Specialities

- given names for components of vector fields:

```
Variable = var_name[cname 1:#dofs cname 2:#dofs ... ]
```

- "internal" Solver can be run as external Procedures (enabling definition of variable names)

```
Procedure = "FlowSolve" "FlowSolver"
```

```
Variable = Flow[Veloc:3 Pres:1]
```

- Residuals of solver variables (e.g., Navier Stokes):

```
Procedure = "FlowSolve" "FlowSolver"
```

```
Variable = Flow[Veloc:3 Pres:1]
```

```
Exported Variable 1 = Flow Loads[Stress Vector:3 CEQ Residual:1]
```

- Solver execution:

```
Exec Solver = {Before Simulation, After Simulation, Never, Always}
```

# Elmer Parallel Version

## ● Pre-processing: ElmerGrid with options:

### Partition by direction:

`-partition 2 2 1 0` First partition elements (default)

`-partition 2 2 1 1` First partition nodes

$$2 \times 2 \times 2 = 8$$

### Partition using METIS:

`-metis n 0` PartMeshNodal (default)

`-metis n 1` PartGraphRecursive

`-metis n 2` PartGraphKway

`-metis n 3` PartGraphVKway

# Elmer Parallel Version

## ● Pre-processing: ElmerGrid with options:

### Partition by direction:

`-partition 2 2 1 0` First partition elements (default)

`-partition 2 2 1 1` First partition nodes

$$2 \times 2 \times 2 = 8$$

### Partition using METIS:

`-metis n 0` PartMeshNodal (default)

`-metis n 1` PartGraphRecursive

`-metis n 2` PartGraphKway

`-metis n 3` PartGraphVKway

## ● Execution: `mpirun -np n ElmerSolver_mpi`

# Elmer Parallel Version

## ● Pre-processing: ElmerGrid with options:

### Partition by direction:

`-partition 2 2 1 0` First partition elements (default)

`-partition 2 2 1 1` First partition nodes

$$2 \times 2 \times 2 = 8$$

### Partition using METIS:

`-metis n 0` PartMeshNodal (default)

`-metis n 1` PartGraphRecursive

`-metis n 2` PartGraphKway

`-metis n 3` PartGraphVKway

## ● Execution: `mpirun -np n ElmerSolver_mpi`

## ● Combining parallel results: in mesh-database directory

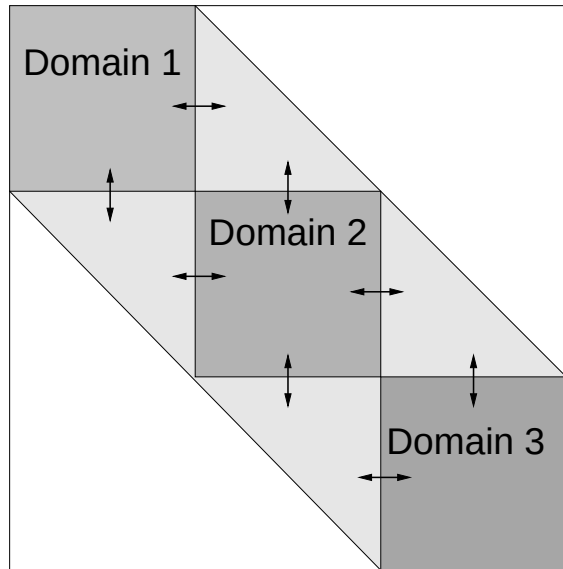
`ElmerGrid 15 3 name`

# Elmer Parallel Version contd.

- need iterative method for linear solver!

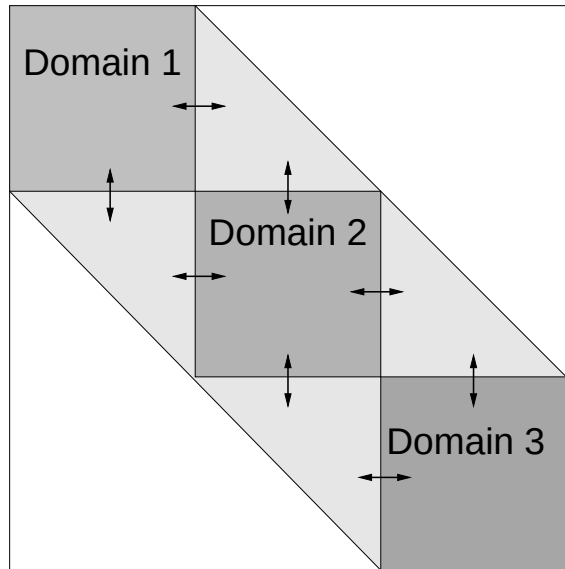
# Elmer Parallel Version contd.

- need iterative method for linear solver!
- standard Krylov subspace in domain decomposition shows different behaviour!



# Elmer Parallel Version contd.

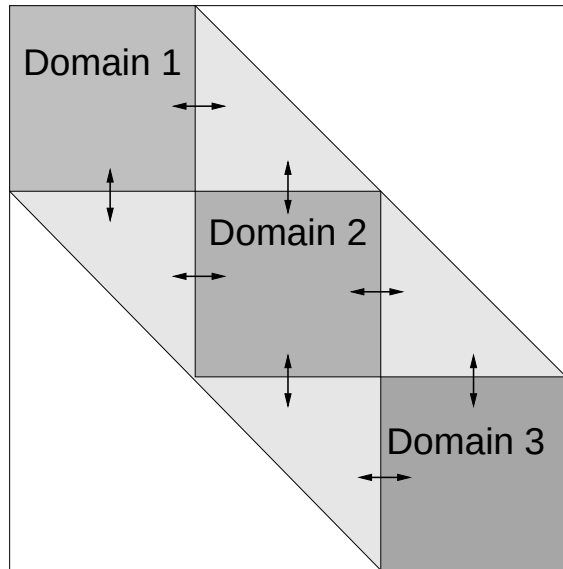
- need iterative method for linear solver!
- standard Krylov subspace in domain decomposition shows different behaviour!



- Linear System Use Hypre = Logical True

# Elmer Parallel Version contd.

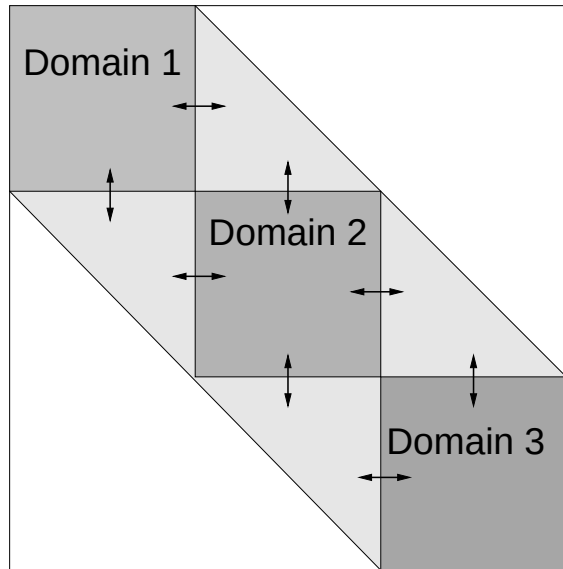
- need iterative method for linear solver!
- standard Krylov subspace in domain decomposition shows different behaviour!



- Linear System Use Hypre = Logical True
- Linear System Preconditioning = ParaSails

# Elmer Parallel Version contd.

- need iterative method for linear solver!
- standard Krylov subspace in domain decomposition shows different behaviour!



- Linear System Use Hypre = Logical True
- Linear System Preconditioning = ParaSails
- ParaSails Threshold, ParaSails Filter, ParaSails Maxlevel, ParaSails Symmetry