CSC

Elmer

Software Development Practices
APIs for Solver and UDF

Peter Råback

ElmerTeam
CSC – IT Center for Science

Tampere Univ. Of Tech.
24.4.2014

Elmer programming languages

- Fortran90 (and newer)
  - ElmerSolver (~240,000 lines of which ~50% in DLLs)
- C++
  - ElmerGUI (~18,000 lines)
  - ElmerSolver (~10,000 lines)
- C
  - ElmerPost (~45,000 lines)
  - ElmerGrid (~30,000 lines)
  - MATC (~11,000 lines)

Elmer libraries

- ElmerSolver
  - Required: Matc, HutIter, Lapack, Blas, Umfpack (GPL)
  - Optional: Arpack, Mumps, Hypre, Pardiso, Trilinos, SuperLU, Cholmod, NetCDF, HDF5, …
- ElmerGUI
  - Required: Qt, ElmerGrid, Netgen
  - Optional: Tetgen, OpenCASCADE, VTK, QVT

Elmer licenses

- ElmerSolver library is published under LGPL
  - Enables linking with all license types
  - It is posible to make a new solver even under propriерity license
  - Note: some optional libraries may constrain this freedom
- Rest of Elmer is published under GPL
  - Derived work must also be under same license (“copyleft”)

Elmer at sourceforge.net

CSC

Home / Browse / Mathematics / Elmer finite element software /

Summary Files Reviews Support Develop Tracker Code

Elmer finite element software

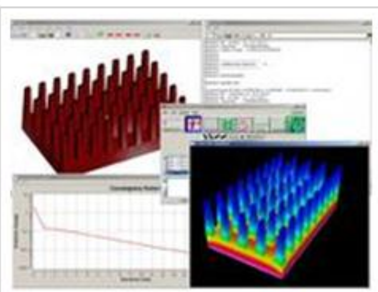
apursula, juhar, juhavierinen, mlsf, mmalinen, raback, sjsillan, tzwinger

54 Recommendations
305 Downloads (This Week)

Download
Elmer-svn5475-2011-12-16.exe

Browse All Files

Tweet 0 +1 0 Like 3



Description

Elmer is a finite element software for numerical solution of partial differential equations and multiphysical problems. It includes models of structural mechanics, fluid dynamics, heat transfer, electromagnetics etc. Elmer home is www.csc.fi/elmer

Elmer finite element software Web Site ›

Obtaining the source code

- Note: elmerfem repository at sf.net was migrated into a new svn server in Jan 2013
- To check-out the code anonymously:

```
svn checkout
svn://svn.code.sf.net/p/elmerfem/code/trun
k elmerfem-code
```

- To check-out the code with a sf.net username:

```
svn checkout --username=raback
svn+ssh://raback@svn.code.sf.net/p/elmerfe
m/code/trunk elmerfem-code
```

- We don’t utilize branches and consider the trunk version to be stable and always backward compatible
  - The things under development may be little bit volatile

Code organization

Directory listing of elmerfem/trunk:

Name	Date modified	Type	
buildtools	25.11.2010 14:33	File folder	
eio	25.11.2010 14:37	File folder	Library for reading the mesh
elmergrid	18.4.2011 23:13	File folder	ElmerGrid
ElmerGUI	25.11.2010 14:37	File folder	ElmerGUI
ElmerGUIlogger	18.4.2011 23:13	File folder	
ElmerGUItester	25.11.2010 14:35	File folder	
elmerparam	25.11.2010 14:34	File folder	ElmerParam
fem	16.9.2011 9:42	File folder	ElmerSolver
front	25.11.2010 14:35	File folder	ElmerFront (obsolite)
hutiter	25.11.2010 14:33	File folder	
matc	25.11.2010 14:33	File folder	MATC library
mathlibs	18.4.2011 23:13	File folder	Basic math libraries
meshgen2d	25.11.2010 14:37	File folder	Mesh2D (almost obsolite)
misc	16.8.2011 13:26	File folder	Miscallenous features
post	25.11.2010 14:34	File folder	ElmerPost
umfpack	25.11.2010 14:34	File folder	Umfpack
utils	25.11.2010 14:35	File folder	Various utilities
LICENSES	25.11.2010 14:37	File	Information on LICENCES

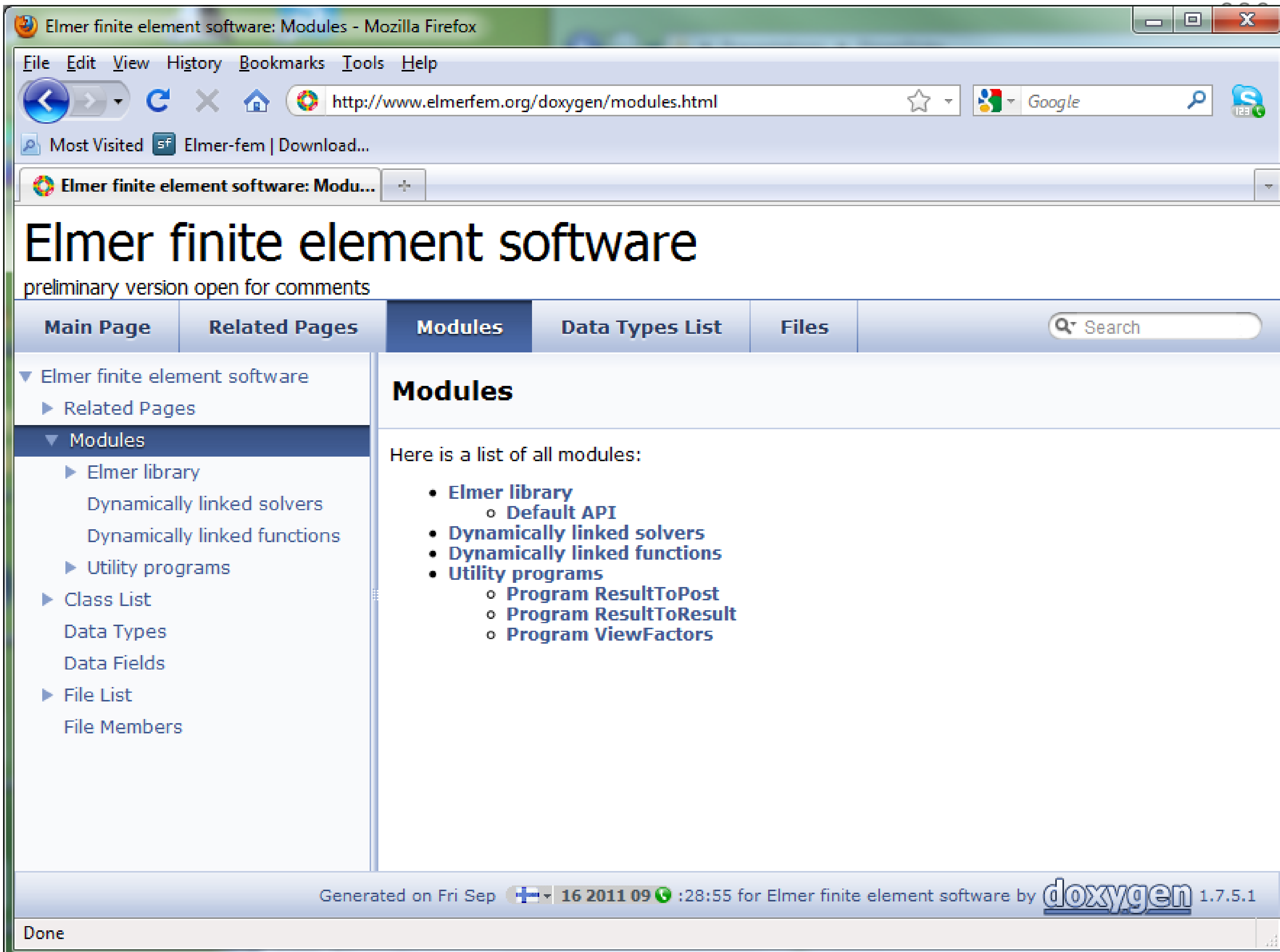
Consistency tests

- Simple shell script to run through the cases + piece of C-code to compare the norm of solutions
- There are about 220 consistency tests (Jan 2013)
  - Located under fem/tests
- Each time a significant commit is made the tests are run with the fresh version
  - Aim: trunk version is a stable version
  - New tests for each major new feature
- The consistency tests provide a good starting point for taking some Solver into use
  - cut-paste from sif file
- Note: the consistency tests have often poor time and space resolution for rapid execution

Consistency tests - example

```
raback@hippu2:/fs/proj1/elmer/raback/elmerfem/fem/tests> ./runtests
$ELMER_HOME undefined, setting it to ../src
test 1 :                 1dtests              [PASSED], CPU time=0.1
test 2 :                 1sttime              [PASSED], CPU time=0.58
test 3 :                 2ndtime              [PASSED], CPU time=1.09
test 4 :              AdvReactDG              [PASSED], CPU time=1.53
test 5 :          BlockLinElast1              [PASSED], CPU time=2.06
test 6 :          BlockLinElast2              [PASSED], CPU time=2.32
test 7 :          BlockLinElast3              [PASSED], CPU time=6.32
test 8 :           BlockPoisson1              [PASSED], CPU time=6.46
test 9 :           BlockPoisson2              [PASSED], CPU time=6.7
test 10 :           BlockPoisson3             [PASSED], CPU time=7.37
test 11 :      CapacitanceMatrix              [PASSED], CPU time=7.66
test 12 :              CavityLid              [PASSED], CPU time=8.46
test 13 :             CavityLid2              [PASSED], CPU time=14.21
test 14 :        ConditionalFlow look at [ConditionalFlow/test.log] for
test 15 :      CoordinateScaling              [PASSED], CPU time=14.34
…
test 189 :                 vortex3d           [PASSED], CPU time=809.12
Tests completed, passed: 188 out of total 189 tests
Cumulative CPU time used in test: 809.12 s
```

Doxygen – WWW documentation



Doxygen – Example in code

- Special comment indicators: !> and <!

```
!------------------------------------------------------------------
!> Subroutine for computing fluxes and gradients of scalar fields.
!> For example, one may compute the the heat flux as the negative grad
!> field multiplied by the heat conductivity.
!> \ingroup Solvers
!------------------------------------------------------------------
SUBROUTINE FluxSolver( Model,Solver,dt,Transient )
!------------------------------------------------------------------
USE CoordinateSystems
USE DefUtils
IMPLICIT NONE
!------------------------------------------------------------------
TYPE(Solver_t) :: Solver   !< Linear & nonlinear equation solver option
TYPE(Model_t) :: Model     !< All model information (mesh, materials, BC
REAL(KIND=dp) :: dt        !< Timestep size for time dependent simulati
LOGICAL :: Transient       !< Steady state or transient simulation
!------------------------------------------------------------------
!    Local variables
!------------------------------------------------------------------
TYPE(ValueList_t),POINTER :: SolverParams
```

Doxygen – Example in WWW

```
subroutine FluxSolver ( TYPE(Model_t)   Model,
                        TYPE(Solver_t)  Solver,
                        REAL(KIND=dp)   dt,
                        LOGICAL         Transient
                      )
```

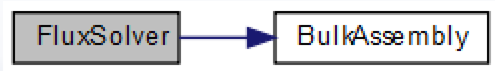
Subroutine for computing fluxes and gradients of scalar fields. For example, one may compute the the heat flux as the negative gradient of temperature field multiplied by the heat conductivity.

Parameters:

Solver	Linear & nonlinear equation solver options
Model	All model information (mesh, materials, BCs, etc...)
dt	Timestep size for time dependent simulations
Transient	Steady state or transient simulation

References **BulkAssembly()**.

Here is the call graph for this function:



Compilation of the whole code

- To compile the whole code see example scripts under www.csc.fi/elmer and www.elmerfem.org

```
#!/bin/sh -f
# Compile Elmer modules and install it
# replace these with your compilers:
export CC=gcc
export CXX=g++
export FC=gfortran
export F77=gfortran

modules="matc umfpack mathlibs elmergrid meshgen2d eio hutiter fem"
for m in $modules; do
  cd $m
  make clean
  ./configure --prefix=/fs/proj1/elmer/raback/elmerbin
  make
  make install
  cd ..
done
```

Compilation of a DLL module

- Applies both to Solvers and User Defined Functions (UDF)
- Assumes that there is a working compile environment that provides ”**`elmerf90`**” script
  - Comes with the Windows installer, and Linux packages
  - Generated automatically when ElmerSolver is compiled

```
elmerf90 MySolver.f90 -o MySolver.so
```

Elmer – High level abstractions

- The quite good success of Elmer as a multiphysics code may be addressed to certain design choices
  - Solver is an asbtract dynamically loaded object
  - Parameter value is an abstract property fecthed from a list
- The abstractions mean that new solvers may be implemented without much need to touch the main library
  - Minimizes need of central planning
  - Several applications fields may live their life quite independently (electromagnetics vs. glaceology)
- MATC – a poor man’s Matlab adds to flexibility as algebraic expressions may be evalueted on-the-fly

Solver as an abstract object

- Solver is an dynamically loaded object (.dll or .so)
  - May be developed and compiled seperately
- Solver utilizes heavily common library utilities
  - Most common ones have interfaces in DefUtils
- Any solver has a handle to all of the data
- Typically a solver solves a weak form of a differential equation
- Currently ~50 different Solvers,
  roughly half presenting physical phenomena
  - No upper limit to the number of Solvers
- Solvers may be active in different domains,
  and even meshes
- The menu structure of each solver in ElmerGUI may be defined by an `.xml` file

Property as an abstract object

- Properties are saved in a list structure by their name
- Namespace of properties is not fixed, they may be introduced in the command file
  - E.g. ”`MyProperty = Real 1.23`” adds a property ”MyProperty” to a list structure related to the solver block
- In code parameters are fetched from the list
  - E.g. ”`val = GetReal( Material,'MyProperty',Found`)” retrieves the above value 1.23 from the list
- A ”`Real`” property may be any of the following
  - Constant value
  - Linear or cubic dependence via table of values
  - Expression given by MATC (MatLab-type command language)
  - User defined functions with arbitrary dependencies
  - Real vector or tensor
- As a result solvers may be weakly coupled without any *a priori* defined manner
- There is a price to pay for the generic approach but usually it is less than 10%
- `SOLVER.KEYWORDS` file may be used to give the types for the keywords in the command file

DefUtils

- DefUtils module includes wrappers to the basic tasks common to standard solvers
  - E.g. ”**`DefaultDirichlet()`**” sets Dirichlet boundary conditions to the given variable of the Solver
  - E.g. ”**`DefaulSolve()`**” solves linear systems with all available direct, iterative and multilevel solvers, both in serial and parallel
- Programming new Solvers and UDFs may usually be done without knowledge of other modules

DefUtils – some functions

Public Member Functions

TYPE(Solver_t) function, pointer	**GetSolver** ()
TYPE(Matrix_t) function, pointer	**GetMatrix** (USolver)
TYPE(Mesh_t) function, pointer	**GetMesh** (USolver)
TYPE(Element_t) function, pointer	**GetCurrentElement** (Element)
INTEGER function	**GetElementIndex** (Element)
INTEGER function	**GetNOFActive** (USolver)
REAL(KIND=dp) function	**GetTime** ()
INTEGER function	**GetTimeStep** ()
INTEGER function	**GetTimeStepInterval** ()
REAL(KIND=dp) function	**GetTimestepSize** ()
REAL(KIND=dp) function	**GetAngularFrequency** (ValueList, Found)
INTEGER function	**GetCoupledIter** ()
INTEGER function	**GetNonlinIter** ()
INTEGER function	**GetNOFBoundaryElements** (UMesh)
subroutine	**GetScalarLocalSolution** (x, name, UElement, USolver, tStep)
subroutine	**GetVectorLocalSolution** (x, name, UElement, USolver, tStep)
INTEGER function	**GetNofEigenModes** (name, USolver)
subroutine	**GetScalarLocalEigenmode** (x, name, UElement, USolver, NoEigen, ComplexPart)
subroutine	**GetVectorLocalEigenmode** (x, name, UElement, USolver, NoEigen, ComplexPart)
CHARACTER(LEN=MAX_NAME_LEN) function	**GetString** (List, Name, Found)
INTEGER function	**GetInteger** (List, Name, Found)
LOGICAL function	**GetLogical** (List, Name, Found)
recursive REAL(KIND=dp) function	**GetConstReal** (List, Name, Found, x, y, z)
recursive REAL(KIND=dp) function	**GetCReal** (List, Name, Found)
recursive REAL(KIND=dp) function, dimension(:), pointer	**GetReal** (List, Name, Found, UElement)

Solver API

```
!------------------------------------------------------
!> Standard API for Solver
!------------------------------------------------------
SUBROUTINE MySolver( Model,Solver,dt,Transient )
!------------------------------------------------------
  USE DefUtils
  IMPLICIT NONE
!------------------------------------------------------
  TYPE(Solver_t) :: Solver  !< Current solver
  TYPE(Model_t) :: Model    !< Handle to all data
  REAL(KIND=dp) :: dt       !< Timestep size
  LOGICAL :: Transient      !< Time-dependent or not
!------------------------------------------------------
  Actual code …
```

Solver API

```
Solver 1
   Equation = "MySolver"
   Procedure = "MyModule" "MySolver"
   …
End
```

- Solver is typically a FEM implementation of a physical equation
- But it could also be an auxiliary solver that does something completely different
- Solver is usually called once for each coupled system iteration

User defined function API

```
!------------------------------------------------------
!> Standard API for UDF
!------------------------------------------------------
FUNCTION MyProperty( Model, n, t ) RESULT(f)
!------------------------------------------------------
  USE DefUtils
  IMPLICIT NONE
!------------------------------------------------------
  TYPE(Model_t) :: Model     !< Handle to all data
  INTEGER :: n               !< Current node
  REAL(KIND=dp) :: t         !< Parameter(s)
  REAL(KIND=dp) :: f         !< Parameter value at node
!------------------------------------------------------
  Actual code …
```

Function API

```
MyProperty = Variable time
   "MyModule" "MyProperty"
```

- User defined function (UDF) typically returns a real valued property at a given point
- It can be located in any section that is used to fetch these values from a list
  - Boundary Condition, Initial Condition, Material,…

Example: Poisson equation

$$-\nabla^2 \phi = \rho$$

- Implemented as an dynamically linked solver
  - Available under tests/1dtests
- Compilation by:
  `Elmerf90 Poisson.f90 -o Poisson.so`
- Execution by:
  `ElmerSolver case.sif`

- The example is ready to go massively parallel and with all a plethora of elementtypes in 1D, 2D and 3D

Poisson equation: code Poisson.f90

```
!------------------------------------------------------------------------------
!> Solve the Poisson equation -\nabla\cdot\nabla \phi = \rho
!------------------------------------------------------------------------------
SUBROUTINE PoissonSolver( Model,Solver,dt,TransientSimulation )
!------------------------------------------------------------------------------
  USE DefUtils
  IMPLICIT NONE
  …

   !Initialize the system and do the assembly:
   !------------------------------------------
   CALL DefaultInitialize()

   active = GetNOFActive()
   DO t=1,active
     Element => GetActiveElement(t)
     n = GetElementNOFNodes()

     LOAD = 0.0d0
     BodyForce => GetBodyForce()
     IF ( ASSOCIATED(BodyForce) ) &
       Load(1:n) = GetReal( BodyForce, 'Source', Found )

     ! Get element local matrix and rhs vector:
     !----------------------------------------
     CALL LocalMatrix(  STIFF, FORCE, LOAD, Element, n )

     ! Update global matrix and rhs vector from local contribs
     !---------------------------------------------------------------
     CALL DefaultUpdateEquations( STIFF, FORCE )
   END DO

   CALL DefaultFinishAssembly()
   CALL DefaultDirichletBCs()
   Norm = DefaultSolve()
```

```
CONTAINS

!------------------------------------------------------------------------------
  SUBROUTINE LocalMatrix(  STIFF, FORCE, LOAD, Element, n )
!------------------------------------------------------------------------------

…

CALL GetElementNodes( Nodes )
    STIFF = 0.0d0
    FORCE = 0.0d0

    ! Numerical integration:
    !----------------------
    IP = GaussPoints( Element )
    DO t=1,IP % n
      ! Basis function values & derivatives at the integration point:
      !--------------------------------------------------------------
      stat = ElementInfo( Element, Nodes, IP % U(t), IP % V(t), &
             IP % W(t),  detJ, Basis, dBasisdx )

      ! The source term at the integration point:
      !------------------------------------------
      LoadAtIP = SUM( Basis(1:n) * LOAD(1:n) )

      ! Finally, the elemental matrix & vector:
      !----------------------------------------
      STIFF(1:n,1:n) = STIFF(1:n,1:n) + IP % s(t) * DetJ * &
          MATMUL( dBasisdx, TRANSPOSE( dBasisdx ) )
      FORCE(1:n) = FORCE(1:n) + IP % s(t) * DetJ * LoadAtIP * Basis(1:n)
    END DO
!------------------------------------------------------------------------------
  END SUBROUTINE LocalMatrix
!------------------------------------------------------------------------------
END SUBROUTINE PoissonSolver
!------------------------------------------------------------------------------
```

Poisson equation: command file case.sif

```
Check Keywords "Warn"

Header
  Mesh DB "." ”mesh"
End

Simulation
  Coordinate System = "Cartesian"
  Simulation Type = Steady State
  Steady State Max Iterations = 50
End

Body 1
  Equation = 1
  Body Force = 1
End

Equation 1
  Active Solvers(1) = 1
End

Solver 1
  Equation = "Poisson"
  Variable = "Potential"
  Variable DOFs = 1
  Procedure = "Poisson" "PoissonSolver"
  Linear System Solver = "Direct”
  Linear System Direct Method = umfpack
  Steady State Convergence Tolerance = 1e-09
End

Body Force 1
  Source = Variable Potential
    Real Procedure "Source" "Source"
End

Boundary Condition 1
  Target Boundaries(2) = 1 2
  Potential = Real 0
End
```

Poisson equation: source term, examples

Constant source:

```
Source = 1.0
```

Source dependeing piecewise linear on x:

```
Source = Variable Coordinate 1
    Real
      0.0 0.0
      1.0 3.0
      2.0 4.0
    End
```

Source depending on x and y:

```
Source = Variable Coordinate
  Real MATC ”sin(2*pi*tx(0))*cos(2*pi(tx(1))”
```

Source depending on anything

```
Source = Variable Coordinate 1
  Procedure ”Source” ”MySource”
```

Poisson equation: ElmerGUI menus

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE edf>
<edf version="1.0" >
  <PDE Name="Poisson" >
    <Name>Poisson</Name>

    <BodyForce>
    <Parameter Widget="Label" > <Name> Properties </Name> </Parameter>
       <Parameter Widget="Edit" >
         <Name> Source </Name>
         <Type> String </Type>
         <Whatis> Give the source term. </Whatis>
      </Parameter>
    </BodyForce>

    <Solver>
      <Parameter Widget="Edit" >
        <Name> Procedure </Name>
        <DefaultValue> "Poisosn" "PoissonSolver" </DefaultValue>
      </Parameter>
     <Parameter Widget="Edit">
        <Name> Variable </Name>
        <DefaultValue> Potential</DefaultValue>
      </Parameter>
     </Solver>

    <BoundaryCondition>
      <Parameter Widget="Label" > <Name> Dirichlet conditions </Name> </Parameter>
      <Parameter  Widget="Edit">
        <Name> Potential </Name>
        <Whatis> Give potential value for this boundary. </Whatis>
      </Parameter>
    </BoundaryCondition>
  </PDE>
</edf>
```

Development tools for ElmerSolver

- Basic use
  - Editor (emacs, vi, notepad++, jEdit,...)
  - elmerf90 script
- Advanced
  - Editor
  - svn client
  - Compiler suite (gfortran, ifort, pathf90, pgf90,...)
  - Documentation tools (Doxygen, LaTeX)
  - Debugger (gdb)
  - Profiling tools
  - ...

Elmer – some best practices

CSC

- Use version control when possible
  - If the code is left to your own local disk, you might as well not write it at all
  - Never fork! (userbase of 1000’s)
- Always make a consistency test for a new feature
  - Always be backward compatible
  - If not, implement a warning to the code
- Maximize the level of abstraction
  - Essential for multiphysics software
  - E.g. any number of physical equations,
    any number of computational meshes,
    any number of physical or numerical parameters – without the need for recompilation