Elmer

Software Development Practices
APIs for Solver and UDF

ElmerTeam

CSC – IT Center for Science, Finland

CSC, 2018

Elmer programming languages

- Fortran90 (and newer)
  - ElmerSolver (~¨300,000 lines of which ~50% in DLLs)
- C++
  - ElmerGUI (~18,000 lines)
  - ElmerSolver (~15,000 lines)
- C
  - ElmerGrid (~30,000 lines)
  - MATC (~11,000 lines)
  - ElmerPost (~45,000 lines)

Tools for Elmer development

- Programming languages
  - Fortran90 (and newer), C, C++
- Compilation & testing
  - Compiler (e.g. gnu), cmake, ctest
- Editing
  - emacs, vi, notepad++,...
- Code hosting (git)
  - https://github.com/ElmerCSC
- Consistency tests
  - Currently more than 500
- Code documentation
  - Doxygen
- Theory documentation

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 possible to make a new solver even under proprierity license
  - Note: some optional libraries may constrain this freedom due to use of GPL licences
- Most other parts of Elmer published under GPL
  - Derived work must also be under same license ("copyleft")
- Proprierity modules linked with ElmerSolver may be freely licensed if they are not derived work
  - Note that you must not violete licences of other libraries

Elmer version control at GitHub

- Elmer source code is hosted at https://github.com/ElmerCSC
- Git offers extreme flexibility
  - Distributed version control system
  - Easy to maintain several development branches
  - Many options and hence also steeper learning curve
  - Developed by Linus Torvalds to host Linux kernel development
- GitHub is a portal providing Git and some additional servives
  - Management of user rights
  - Controlling pull requests

Cmake build system

- Elmer currently uses cmake for building since 2015
- Cmake offers several advantages (over gnu autotools)
  - Enables cross compilation for diffirent platforms (e.g. Intel MICs)
  - More standardizes installation scripts
  - Straight-forward package creation for many systems (using cpack)
  - Great testing utility with ctest – now also in parallel
- Transition to cmake required significant code changes
  - ISO C-bindings & many changes in APIs
  - Backward compatibility in compilation lost

Compiling fresh Elmer source from GitHub

```
# clone the git repository.
$ git clone https://www.github.com/ElmerCSC/elmerfem

# Switch to devel branch (currently the default branch)
$ cd elmerfem
$ git checkout devel
$ cd ..

# create build directory
$ mkdir build
$ cd build
```

```
$ cmake -DWITH_ELMERGUI:BOOL=FALSE -DWITH_MPI:BOOL=FALSE -DCMAKE_INSTALL_PREFIX=../install ../elmerfem
```

```
$ cmake <flags>
# You can tune the compilation parameters graphically with $ ccmake or $cmake-gui .

$ make install
# or alternatively compile in parallel (4 procs) $ make -j4 install
```

Consistency tests

- Utilize ctest system to run a set of Elmer cases
  - Upon success each case writes 1 to file TEST.PASSED, and on failure 0, respectively
- There are more than 580 consistency tests (May 2018)
  - Located under fem/tests
- Each time a significant commit is made the tests are run with the fresh version
  - Aim: even devel version is a stable
  - 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

Executing the consistency tests of Elmer

```
>ctest -j4 -LE elmerice
      Start 143: mgdyn_torus_harmonic
        Start 304: ThermalActuator
        Start 344: RotatingBCMagnetoDynamicsGeneric
  1/310 Test #344: RotatingBCMagnetoDynamicsGeneric ...   Passed   43.18 sec
        Start 293: mgdyn_lamstack_lowfreq_harmonic
  2/310 Test #304: ThermalActuator ....................   Passed   59.78 sec
        Start 222: mgdyn_transient_loss
  3/310 Test #293: mgdyn_lamstack_lowfreq_harmonic ....   Passed   21.80 sec
        Start 322: mgdyn_bh

…

308/310 Test  #46: CoupledPoisson7 ....................   Passed    0.38 sec
309/310 Test #212: CoordinateScaling ..................   Passed    0.38 sec
        Start  54: RotatingBCPoisson3DSymmSkev
310/310 Test  #54: RotatingBCPoisson3DSymmSkev ........   Passed    6.34 sec

100% tests passed, 0 tests failed out of 310

Total Test time (real) = 365.62 sec
```

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
```

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,…
- Note: function is called for all nodes (or gauss points) of all elements
  - Save constly initializations!

UDF Example: sinusoidal heat source

```
FUNCTION MySource( Model, n, t ) RESULT( f )
  USE DefUtils
  IMPLICIT NONE

  TYPE(Model_t) :: Model
  INTEGER :: n
  REAL(KIND=dp) :: t, f
  REAL(KIND=dp), PARAMETER :: a=1.23, w=4.56

  f = a*sin(w*t)

END FUNCTION MySource
```

```
Body Force 1
  Name = ”Heating”
  Heat Source = Variable time
    Real Procedure ”MyModule” ”Sinus”
End

!same function using MATC
Body Force 1
  $a=1.23
  $w=4.56
  Heat Source = Variable time
    Real MATC ”a*sin(w*t)
End
```

UDF Example: sinusoidal heat source with SIF control

```
FUNCTION MySource( Model, n, t ) RESULT( f )
  USE DefUtils
  IMPLICIT NONE

  TYPE(Model_t) :: Model
  INTEGER :: n
  REAL(KIND=dp) :: t, f
  REAL(KIND=dp) :: a=1.23, w=4.56
  LOGICAL :: Visited = .FALSE.
  SAVE a, w, VIsited

  IF(.NOT. Visited ) THEN
    a = ListGetConstReal( Model % Simulation,’My Amplitude’)
    w = ListGetConstReal( Model % Simulation,’My Angular Velocity’)
    Visited = .TRUE.
  END IF
  f = a*sin(w*t)
END FUNCTION MySource
```

```
Simulation
  …
  My Amplitude = Real 1.23
  My Angular Velocity = Real 4.56
End

Body Force 1
  Name = ”Heating”
  Heat Source = Variable time
    Real Procedure ”MyModule” ”Sinus”
End
```

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

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 ~60 different Solvers,
  roughly half presenting physical phenomena
  - No upper limit to the number of Solvers
  - Often cases include ~10 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

Code structure

- Elmer code structure has evolved over the years
  - There has been no major restructuring operations
- Ufortunately there is no optimal hierarchy and the number of subroutines is rather large
  - ElmerSolver library consists of more than ~40 modules
  - There are all-in-all around 1050 SUBROUTINES and 650 FUNCTIONS (both internal and external)
- To ease the learning curve the most important routines for basic use have been collected into module DefUtils.F90

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. ”**`DefaultSolve()`**” 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)

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>
```

Elmer – some best practices

- Use version control when
  - If the code is left to your own local disk, you might as well not write it at all
  - Merge often to the upstream, rather not fork
- 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