Basic programming with Elmer

Mikko Lyly

CSC

20.10.2009

Basic programming with Elmer

ElmerSolver is written in Fortan 90. It consists of approximately 300 000 lines of source code (including all 3rd party libraries).

The base FEM-code has a modular structure, in the sense that it is possible to build additional components without recompiling the whole FEM package.

Basic concepts

Typical cases in which programming is needed are the following:

- complicated boundary conditions or material parameters need to be evaluated pointwise

- new finite element methods need to be implemented for specific purposes

Basic concepts (example 1)

Suppose that we want to define boundary condition #1 on boundary part #5 as follows:

$$\text{Temperature}(x,y,t) = t * \sin(x) * \cos(y)$$

This can be accomplished by introducing the following boundary condition block in the Solver Input File:

```
Boundary condition 1
  Target bondaries(1) = 5
  Temperature = Variable Time
    Real Procedure ”MyDll” ”MyFunction”
End
```

Basic Conceps (example 1)

Here ”MyDll” is the name of a shared library (loaded once and for all when Solver starts), and ”MyFunction” is a library function to execute. The function ”MyFunction” will be called automatically by ElmerSolver for each node.

Let ”MyDll.f90” be the source file for ”MyDll”. The source should have the following structure:

Basic Concepts (example 1)

```
! File MyDll.f90

Function MyFunction( Model, n, time ) RESULT( temp )
  Use List
  TYPE(Model_t) :: Model      ! defined in Types.f90
  INTEGER :: n                ! node number
  REAL(KIND=dp) :: time       ! Variable Time in sif
  REAL(KIND=dp) :: temp       ! Temperature in sif

  !...compute temp ...

END FUNCTION MyFunction
```

Basic Conceps (example 1)

In order to compute sin(x) and cos(y), we will have to query coordinates from the Model structure:

```
REAL(KIND=dp) :: X, Y

X = Model % Nodes % x(n)
Y = Model % Nodes % y(n)
```

Time comes as an argument in the function call. So, it remains to compute

```
temp = time * SIN(X) * COS(Y)
```

and we are done. Summing up, the full code is

Basic Concepts (example 1)

```
! File MyDll.f90

Function MyFunction( Model, n, time ) RESULT( temp )
  Use Lists
  TYPE(Model_t) :: Model       ! defined in Types.f90
  INTEGER :: n                 ! node number
  REAL(KIND=dp) :: time        ! Variable Time in sif
  REAL(KIND=dp) :: temp        ! Temperature in sif
  Real :: X, Y
  X = Model % Nodes % x(n)
  Y = Model % Nodes % y(n)
  temp = time * SIN(X) * COS(Y)
END FUNCTION MyFunction
```

Basic Concepts (example 1)

It remains to compile the function for ElmerSolver:

Linux:

```
$ elmerf90 MyDll.f90 -o MyDll.so
```

Windows:

```
> elmerf90 MyDll.f90 -o MyDll.dll
```

The users of ElmerGUI may compile the function directly from menu (Run → Compiler...)

Basic Concepts

Local arrays (if needed) should be declared ALLOCATABLE, SAVEd, and ALLOCATEd once and for all:

```
LOGICAL :: FirstTime = .TRUE.
REAL(KIND=dp), ALLOCATABLE :: MyArray(:)
SAVE MyArray, FirstTime
...
IF( FirstTime ) THEN
   ALLOCATE( MyArray( 100 ) )
   FirstTime = .FALSE.
END IF
```

Otherwise, arrays will be allocated everytime the function is entered, and deallocated when out of scope (=penalty in speed, memory usage).

Basic Conceps (custom solver)

Another important case in which programming is needed is when the user wants to implement a custom Solver for his/her equation. In the Solver Input File the Solver block related a custom equation should be defined as follows:

```
Solver 1
  Variable = String MyVariable
  Variable DOFs = Integer 1
  Procedure = ”MyDll” ”MyRoutine”
  ...
End
```

Basic Concepts (custom solver)

All custom solvers have the following fixed calling convention:

```
SUBROUTINE MySolver( Model,Solver,dt,TransientSimulation )
  USE DefUtils
  TYPE(Model_t) :: Model
  TYPE(Solver_t) :: Solver
  REAL(KIND=dp) :: dt
  LOGICAL :: TransientSimulation
  ...
END SUBROUTINE MySolver
```

Basic Concepts (custom solver)

Again, all local arrays should be declared ALLOCATABLE, SAVEd, and ALLOCATEd once and for all. For example:

```
LOGICAL :: FirstTime = .TRUE.
REAL(KIND=dp), ALLOCATABLE :: Stiff(:,:), Force(:)
SAVE Stiff, Force
...
IF( FirstTime ) THEN
  ALLOCATE( Stiff(12, 12), Force( 12 ) )
  FirstTime = .FALSE.
END IF
```

Basic Concepts (custom solver)

The fundamental task of a custom solver is to form the global stiffness matrix related to a PDE. This is done by performing a loop over elements, computing the local stiffness matrices, and by assembling the global matrix:

```
INTEGER :: t
TYPE(Element_t), POINTER :: Element
...
DO t = 1, GetNOFActive()
  Element => GetActiveElement(t)
  ...
END END
```

The functions GetNofActive() and GetActiveElement() are defined in DefUtils.f90.

Basic Concepts (custom solver)

The type Element_t provides useful data for individual elements, for example:

```
Element % ElementIndex      ! index of the element
Element % BodyId            ! index of the domain
Element % NumberofNodes     ! number of nodes
Element % ElementCode       ! type of the element
Element % hK                ! size of the element
...
```

For more details, see fem/src/Types.f90

Basic Concepts (custom solver)

Next, the user usually wants to query the material parameters and loads for a given element. This can be done by calling the function GetReal() defined in DefUtils:

```
TYPE(ValueList_t), POINTER :: Material
LOGICAL :: Found
INTEGER :: N
REAL(KIND=dp), ALLOCATABLE :: MatValues(:)
…
  Material => GetMaterial( Element )
  N = GetElementNofNodes( Element )
  MatValues(1:N) = GetReal( Material, 'MyName', Found)
```

The above will query nodal values of MyName from the material block of the Solver Input File.

Basic Concepts (custom solver)

To be a little bit more rigorous, the user should perform some tests and error checks to avoid problems:

```
Material => GetMaterial( Element )
IF( ASSOCIATED( Material ) ) THEN
  N = GetElementNofNodes( Element )
  MatValues(1:N) = GetReal( Material, 'MyName', Found)
  IF( .NOT.Found ) THEN
     ! Handle the missing material param appropriately
  END IF
END IF
```

Basic Concepts (custom solver)

Similarly, we could query the nodal values of a load from the Body Force block of the SIF for example as follows:

```
TYPE(ValueList_t), POINTER :: BodyForce
LOGICAL :: Found
INTEGER :: N
REAL(KIND=dp), ALLOCATABLE :: BFValues(:)
...
  BodyForce => GetBodyForce( Element )
  N = GetElementNofNodes( Element )
  BFValues(1:N) = GetReal( BodyForce, 'Fx', Found)
```

Basic Concepts (custom solver)

Once the parameters and loads are at hand, the next step is to integrate the local stiffness matrix. Usually, this is done in a separate subroutine contained by the solver subroutine:

```
SUBROUTINE MySolver(Model, Solver, dt, TransientSimulation)
  USE DefUtils
  …
  DO t = 1, GetNOFActive()
    Element => GetActiveElement(t)
    CALL LocalStiff( Stiff, Force, Element, n )
    CALL DefaultUpdateEquations( STIFF, FORCE )
  END DO

CONTAINS
  SUBROUTINE LocalStiffness( Stiff, Force, Element, n )
  ...
  END SUBROUTINE LocalStiffness
END SUBROUTINE MySolver
```

Basic Concepts (custom solver)

To be a little more specific, the subroutine for computing local entities has the following basic declarations:

```
SUBROUTINE LocalStiffness( Stiff, Force, Element, n )
  REAL(KIND=dp) :: Stiff(:,:), Force(:)
  TYPE(Element_t) :: Element
  INTEGER :: n
  REAL(KIND=dp) :: Basis(n), dBasisdx(n,3), DetJ
  LOGICAL :: Stat
  INTEGER :: t
  TYPE(GaussIntegrationPoints_t) :: IP
  TYPE(Nodes_t) :: Nodes
  SAVE Nodes
  ...
END SUBROUTINE LocalStiffness
```

Basic Concepts (custom solver)

The first thing to do, before anything else, is to query the node points for the Element under work, and reset the local matrix and vector:

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

Node coordinates are needed to contruct the mapping from the reference element to the actual geometry.

Basic Concepts (custom solver)

In order to evaluate the local stiffness related to a PDE, we will first have to select a quadrature for numerically evaluating integrals:

```
IP = GaussPoints( Element )
```

The variable IP is of the type GaussIntergarionPoints_t :

```
IP % n      ! number of integration points
IP % U(t)   ! U-coordinate for point n
IP % V(t)   ! V-coordinate for point n
IP % W(t)   ! W-coordinate for point n
IP % S(t)   ! weight for point n
```

Basic Concepts (custom solver)

The integration loop is then the following:

```
DO t = 1, IP % n
   stat = ElementInfo( Element, Nodes, IP % U(t),
   IP % V(t), IP % W(t), detJ, Basis, dBasisdx )
   …
END DO
```

The function ElementInfo returns the basis functions and their gradients in the integration point.

Basic Concepts (custom solver)

Given the basis and their gradients, it remains to evaluate the inner products related to the PDE. For the Poisson equation, for example, we do:

```
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 * 1.0 * Basis
```

In the above, we computed the load for f=1.

Basic Concepts (custom solver)

Let us finally tune the local subroutine by passing the nodal values of material parameters to it (similar adjustments needed for the nodal load):

```
CALL LocalStiff( Stiff, Force, Element, MatValues, n )

SUBROUTINE LocalStiffness(Stiff,Force,Element,MatValues, n)
  REAL(KIND=dp) :: MatValues(:)
  REAL(KIND=dp) :: MatValueAtIP
  …
  DO t = 1, IP % n
    Stat = ElementInfo(...)
    MatValueAtIP = SUM( Basis(1:n) * MatValues(1:n) )
    …
    STIFF(1:n, 1:n) = STIFF(1:n,1:n) + IP % s(t) * DetJ * &
    MatValueAtIP * MATMUL( dBasisdx,TRANSPOSE( dBasisdx ) )
    ...
  END DO
```

Basic Concepts (custom solver)

Once the loop over elements has finished, we have the global matrix and vector at hand. It remains to finalize the assembly and actually solve the problem, and we are done:

```
DO t = 1, GetNOFActive()
  Element => GetActiveElement(t)
  CALL LocalStiff( Stiff, Force, Element, MyValues, n )
  CALL DefaultUpdateEquations( STIFF, FORCE )
END END

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

Basic Concepts (custom solver)

A full example of programming a custom solver is provided in the test case Poisson1D. The compilation command is:

Linux:

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

Windows:

```
> elmerf90 MySolver.f90 -o MySolver.dll
```

Basic Concepts

The programming interfce of Elmer has been documented in the SolverManual. A good reference for basic features is also the test set in fem/tests.

Most of the data structures are undocumented, but the source files Types.f90 and DefUtils.f90 should be more or less self explalatory.