Elmer Programmer's Tutorial

Mikko Lyly

CSC – IT Center for Science

2010–2011

Elmer Programmer's Tutorial

About this document

The Elmer Programmer's Tutorials is part of the documentation of Elmer finite element software. It gives examples on how to carry out simple coding tasks using the high-level routines from Elmer library.

The present manual corresponds to Elmer software version 7.0. Latest documentations and program versions of Elmer are available (or links are provided) at `http://www.csc.fi/elmer`.

Copyright information

Contents

Chapter 1

User defined functions

1.1 Calling convention

All user defined functions that implement e.g. a material parameter, body force, or a boundary condition, are written in Fortran90 with the following calling convention:

```
!------------------------------------------------------------------------------
!> File: MyLibrary.f90
!> Written by: ML, 5 May 2010
!> Modified by: -
!------------------------------------------------------------------------------
FUNCTION MyFunction(Model, n, f) RESULT(g)
   USE DefUtils
   TYPE(Model_t) :: Model
   INTEGER :: n
   REAL(KIND=dp) :: f, g

   ! code

END FUNCTION MyFunction
```

The function is called automatically by ElmerSolver for each node index `n`, when activated from the Solver Input File e.g. as follows:

```
Material 1
  MyParameter = Variable Time
    Real Procedure "MyLibrary" "MyFunction"
End
```

In this case, the value of time will be passed to the function in variable `f`. The function then returns the value of the material parameter in variable `h`.

The type `Model_t` is declared and defined in the source file `Types.src` also referenced by `DefUtils.src`. It contains the mesh and all model data specified in the Solver Input File. As an example, the coordinates of node `n` are obtained from `Model` as follows:

```
  REAL(KIND=dp) :: x, y, z
  x = Model % Nodes % x(n)
  y = Model % Nodes & y(n)
  z = Model % Nodes % z(n)
```

If the value of the return value depends on a specific function (for example a temperature dependent heat conducivity), we can fetch the nodal value of that function by using the DefUtils-subtouines (more details to follow in the next section):

```
  TYPE(Variable_t), POINTER :: TemperatureVariable
  REAL(KIND=dp) :: NodalTemperature
  INTEGER :: DofIndex
  TemperatureVariable => VariableGet(Model % Variables, 'Temperature')
  DofIndex = TemperatureVariable % Perm(n)
  NodalTemperature = TemperatureVariable % Values(dofIndex)
  ! Compute heat conductivity from NodalTemperature, k=k(T)
```

1.2 Compilation

The function is compiled into a shared library (Unix-like systems) or into a dll (Windows) by using the default compiler wrapper `elmerf90` (here and in the sequel, $ stands for the command prompt of a bash shell (Unix) and > is input sign of the Command Prompt in Windows):

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

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

Chapter 2

User defined solvers

2.1 Calling convention

All user defined subroutines that implement a custom solver are written in Fortran90 with the following calling convention:

```
!------------------------------------------------------------------------------
! File: MySolver.f90
! Written by: ML, 5 May 2010
! Modified by: -
!------------------------------------------------------------------------------
SUBROUTINE MySolver(Model, Solver, dt, Transient)
  Use DefUtils
  IMPLICIT NONE
  TYPE(Solver_t) :: Solver
  TYPE(Model_t) :: Model
  REAL(KIND=dp) :: dt
  LOGICAL :: Transient

  ! User defined code

END MySolver
```

The types `Solver_t` and `Model_t` are defined in the source file `Types.src`.

2.2 Compilation

The subroutine is compiled into a shared library like a user defined function by using the compiler wrapper `elmerf90`:

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

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

2.3 Solver Input File

The user defined solver is called automatically by ElmerSolver when an appropriate Solver-block is found from the Solver Input File:

```
Solver 1
  Procedure = "MyLibrary" "MySolver"
  ...
End
```

Chapter 3

Reading data from SIF

In this chapter the flow of information from the command file is described. The file is also known as Solver Input File, or sif file. The relevant functions and subroutines are defined in DefUtils.src.

3.1 Reading constant scalars

For reading constant valued scalars the following function is used

```
RECURSIVE FUNCTION GetConstReal(List, Name, Found) RESULT(Value)
  TYPE(ValueList_t), POINTER : List
  CHARACTER(LEN=*) :: Name
  LOGICAL, OPTIONAL :: Found
  REAL(KIND=dp) :: Value
```

Solver Input File:

```
Constants
  MyConstant = Real 123.456
End
```

You may not thet here the type `Real` is defined. The type of fixed keywords are usually defined in file `SOLVER.KEYWORDS` in the `bin` directory. Also the user may create a local copy of the file introducing new variables there.
Code (sif1/MyLibrary.f90):

```
SUBROUTINE MySolver(Model, Solver, dt, Transient)
  USE DefUtils
  IMPLICIT NONE
  TYPE(Solver_t) :: Solver
  TYPE(Model_t) :: Model
  REAL(KIND=dp) :: dt
  LOGICAL :: Transient

  ! Read constant scalar from Constants-block:
  !--------------------------------------------
  REAL(KIND=dp) :: MyConstant
  LOGICAL :: Found

  MyConstant = GetConstReal(Model % Constants, "MyConstant", Found)
  IF(.NOT.Found) CALL Fatal("MySolver", "Unable to find MyConstant")
  PRINT *, "MyConstant =", MyConstant

END SUBROUTINE MySolver
```

Output:

```
 MyConstant =   123.45600000
```

3.2 Reading constant vectors

For reading constant valued vectors or matrices the following function is used

```
RECURSIVE SUBROUTINE GetConstRealArray(List, Value, Name, Found)
  TYPE(ValueList_t), POINTER : List
  CHARACTER(LEN=*) :: Name
  LOGICAL, OPTIONAL :: Found
  REAL(KIND=dp), POINTER :: Value(:,:)
```

Solver Input File:

```
Solver 1
  MyVector(3) = Real 1.2 3.4 5.6
End
```

Code (sif2/MyLibrary.f90)

```
SUBROUTINE MySolver(Model, Solver, dt, Transient)
  Use DefUtils
  IMPLICIT NONE
  TYPE(Solver_t) :: Solver
  TYPE(Model_t) :: Model
  REAL(KIND=dp) :: dt
  LOGICAL :: Transient

  ! Read constant vector from Solver-block:
  !----------------------------------------
  REAL(KIND=dp), POINTER :: MyVector(:,:)
  LOGICAL :: Found

  CALL GetConstRealArray(Solver % Values, MyVector, "MyVector", Found)
  IF(.NOT.Found) CALL Fatal("MySolver", "Unable to find MyVector")
  PRINT *, "MyVector =", MyVector(:,1)

END SUBROUTINE MySolver
```

Output:

```
 MyVector =   1.2000000000        3.4000000000        5.6000000000
```

3.3 Reading constant matrices

Solver Input File:

```
Material 1
  MyMatrix(2,3) = Real 11 12 13                        21 22 23
```

Code (sif3/MyLibrary.f90):

```
SUBROUTINE MySolver(Model, Solver, dt, Transient)
  Use DefUtils
  IMPLICIT NONE
  TYPE(Solver_t) :: Solver
  TYPE(Model_t) :: Model
  REAL(KIND=dp) :: dt
  LOGICAL :: Transient

  ! Read constant matrix from Material-block
  !-----------------------------------------
  REAL(KIND=dp), POINTER :: MyMatrix(:,:)
  LOGICAL :: Found
  TYPE(ValueList_t), POINTER :: Material

  Material => Model % Materials(1) % Values
  CALL GetConstRealArray(Material, MyMatrix, "MyMatrix", Found)
  IF(.NOT.Found) CALL Fatal("MySolver", "Unable to find MyMatrix")
  PRINT *, "Size of MyMatrix =", SIZE(MyMatrix,1), "x", SIZE(MyMatrix,2)
  PRINT *, "MyMatrix(1,:) =", MyMatrix(1,:)
  PRINT *, "MyMatrix(2,:) =", MyMatrix(2,:)

END SUBROUTINE MySolver
```

Output:

```
 Size of MyMatrix =           2 x           3
 MyMatrix(1,:) =   11.000000000       12.000000000       13.000000000
 MyMatrix(2,:) =   21.000000000       22.000000000       23.000000000
```

Chapter 4

Managing variables

In this chapter the treatment of variables is presented.

4.1 Handle to variables

You can access your global solution vector of your finite element subroutine. The following is limited to the field variable that is being solved for:

```
TYPE(Variable_t), POINTER :: MyVariable
REAL(KIND=dp), POINTER :: MyVector(:)
INTEGER, POINTER :: MyPermutation(:)
...
MyVariable => Solver % Variable
MyVector => MyVariable % Values
MyPermutation => MyVariable % Perm
```

Also any other variable may be accessed by its name and thereafter be treated as the default variable. For example

```
Mesh => GetMesh()
MyVariable => VariableGet( Mesh,'ExtVariable')
IF( .NOT. ASSOCIATED ( MyVariable ) ) THEN
  CALL Fatal('MySolver','Could not find variable > ExtVariable < ')
END IF
```

If you want to set all values of the vector to a constant value that would be done simply with

```
MyVector = 123.456
```

4.2 Permutation vector of variable

The integer component `Var % Perm` tells the mapping between physical nodes and field variables. It is zero there where the field variable is not active. The numbering of the non-zero entries must use all integerers starting from 1. Usually the numbering is determined by bandwidth optimization which is always on by default. You can turn the optimization off by adding the line `Bandwidth optimization = FALSE` in the Solver-section of your SIF. In this case the permutation vector `MyPermutation` becomes the identity map.
In the case of a scalar field, you can then set the value of the field e.g. in node 3 as

```
MyVector(MyPermutation(3)) = 123.456
```

Some field variables do not have the Permulation defined and then `MyPermutation` would not be associated. For example, the coordinates are available as field variables `Coordinate 1`, `Coordinate 2` and `Coordinate 3` without the permutation vector.
For example, getting the field variable corresponding to coordinate x could be done either as

```
x = Mesh % Nodes % x( node )
```

or

```
MyVariable => VariableGet( Mesh,'Coordinate 1')
x = MyVariable % Values( node )
```

The alternative way of accessing the coordinates is important since that enables that the same dependency features may be used for true field variables, as well as for coordinates.

4.3 Vector valued field variables

The field variable may also have vector values at each node. If the primary field name is `VarName` then the individual components are by default referred to by their component indexes `VarName i`. The vector valued field are ordered so that for each node the components follow each other.

For example, assume that we want to retrieve the three components of a displacement vector. This could be done as follows:

```
...
MyVariable => GetVariable( Mesh % Variables,'Displacement')
MyVector => MyVariable % Values
MyPermutation => MyVariable % Perm
MyDofs = MyVariable % Dofs

j = MyPermutation(node)
IF( j /= 0 ) ) THEN
  ux = MyVector( Dofs * (j-1)+1 )
  IF( Dofs >= 2 ) uy = MyVector( Dofs * (j-1)+2 )
  IF( Dofs >= 3 ) uz = MyVector( Dofs * (j-1)+3 )
END IF
```

4.4 Global variables

Global variables may be treated similarly as field variables. However, they have no reference to nodes. Examples of global variables are `time, timestep size, nonlin iter` and `coupled iter`.

A good indicator that a variable is global is that its size is equal to the number of i.e. the following condition is true

```
SIZE( MyVector ) == MyDofs
```

4.5 Creating variables

Within the code variables may be created by command `VariabelAddVector`.

In the command file a variable may be created with keyword `Exported Variable i`. It takes also parameters such as `-dofs` and `-global`. So the following expression would create a global variable with 5 degrees of freedom.

```
Exported Variable 1 = -global -dofs 5 MyGlobals
```

Chapter 5

Mesh files

Elmer mesh is defined by a selection of files: `mesh.header, mesh.nodes, mesh.elements` and `mesh.boundary`. In parallel runs there will also be file `mesh.shared`.

The mesh files may be created by ElmerGUI using some of its built-in mesh generators. By ElmerGrid using its native format or import utilities. If the user has his own mesh generator writing a parser to Elmer format will not be a mission impossible.

5.1 Creating mesh files manually

To understand what the mesh file looks like we present a toy mesh. It consists of 6 nodes defined by their (x,y,z) coordinates, 4 linear triangles (Elmer type 303) and 2 different boundaries.
mesh.nodes

```
1 -1 0.0 0.0 0.0
2 -1 0.0 -1.0 0.0
3 -1 1.0 -1.0 0.0
4 -1 1.0 1.0 0.0
5 -1 -1.0 1.0 0.0
6 -1 -1.0 0.0 0.0
```

mesh.elements

```
1 1 303 1 2 3
2 1 303 1 3 4
3 1 303 1 4 5
4 1 303 1 5 6
```

mesh.boundary

```
1 1 1 0 202 1 2
2 1 1 0 202 2 3
3 1 2 0 202 3 4
4 2 3 0 202 4 5
5 2 4 0 202 5 6
6 2 4 0 202 6 1
```

mesh.header

```
6 4 6
2
202 6
303 4
```

Chapter 6

Partial Differential Equations

6.1 Model problem

In this section, we will consider the boundary value problem

$$-\Delta u = f \quad \text{in } \Omega,$$

$$u = 0 \quad \text{on } \partial\Omega,$$

where $\Omega \subset R^d$ is a smooth bounded domain ($d = 1, 2, 3$) and $f = 1$.

The problem can be written as

$$\frac{1}{2}\int_\Omega |\nabla u|^2 \, d\Omega - \int_\Omega fu \, d\Omega = \min!$$

where the minimum is taken over all sufficiently smooth functions that satisfy the kinematical boundary conditions on $\partial\Omega$.

6.2 FEM

The Galerkin FEM for the problem is obtained by dividing Ω into finite elements and by introducing a set of mesh dependent basis functions $\{\phi_1, \phi_2, \ldots, \phi_n\}$. The approximate solution is written as a linear combination of the basis and detemined from the condition that it minimizes the energy:

$$u_n = \sum_{i=1}^{n} \phi_i u_i \quad (u_i \in R)$$

and

$$\frac{1}{2}\int_\Omega |\nabla u_n|^2 \, d\Omega - \int_\Omega fu_n \, d\Omega = \min!$$

The solution satisfies

$$\sum_{j=1}^{n} A_{ij} u_j = f_i, \quad i = 1, 2, \ldots, n,$$

with

$$A_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, d\Omega$$

and

$$f_i = \int_\Omega f\phi_i \, d\Omega.$$

In practice, the coefficients A_{ij} are computed by summing over the elements:

$$A_{ij} = \sum_E A_{ij}^E$$

with

$$A_{ij}^E = \int_E \nabla\phi_i \cdot \nabla\phi_j \, d\Omega$$

The integrals over the elements are evaluated through a mapping $f_E : \hat{E} \to E$, where $\hat{E}$ is a fixed reference element:

$$A_{ij}^E = \int_{\hat{E}} \nabla\phi_i \cdot \nabla\phi_j \, |J_E| \, d\hat{\Omega}$$

where $|J_E|$ is the determinant of the Jacobian matrix of f_E. In most cases, f_E is either an affine or an isoparametric map from the unit triangle, square, tetrahedron, hexahedron etc., into the actual element.

Finally, the integral over the reference element is computed numerically with an appropriate quadrature. Elmer uses the Gauss-quadrature by deault, as most of the FE-codes:

$$A_{ij}^E = \sum_{k=1}^{N} \nabla\phi_i(\xi_k) \cdot \nabla\phi_j(\xi_k) \, w_k \, |J_E(\xi_k)|$$

where ξ_k is the integration point and w_k is the integration weight.

So, the system matrices and vectors of the FEM are formed by implementing a loop over the elements, by computing the local matrices and vectors with an appropriate quadrature, and by assembling the global system from the local contributions.

6.3 Implementation

Let us next implement the method in Elmer and write a user defined subroutine for the Poisson equation. To begin with, let us allocate memory for the local matrices and vectors. This is done once and for all in the beginning of the subroutine:

```
INTEGER :: N
TYPE(Mesh_t), POINTER :: Mesh
LOGICAL :: AllocationsDone = .FALSE.
REAL(KIND=dp), ALLOCATABLE :: Matrix(:,:), Vector(:)
SAVE AllocationsDone, LocalMatrix, LocalVector

IF(.NOT.AllocationsDone) THEN
   Mesh => GetMesh(Solver)
   N = Mesh % MaxElementNodes
   ALLOCATE(Matrix(N,N))
   ALLOCATE(Vector(N))
END IF
```

The next step is to implement a loop over all active elements, call a subroutine that computes the local matrices and vectors (to be specified later), and assemble the global system by using the DefUtils subroutine `DefaultUpdateEquations()`:

```
INTEGER :: i
TYPE(Element_t), POINTER :: Element

DO i = 1, GetNOFActive(Solver)
   Element => GetActiveElement(i)
   N = GetElementNOFNodes(Element)
   CALL ComputeLocal(Element, N, Matrix, Vector)
   CALL DefaultUpdateEquations(Matrix, Vector, Element)
END DO
```

The assembly is finalized by calling the DefUtils subroutine `DefaultFinishAssembly()`. Dirichlet boundary conditions are set by calling the subroutine `DefaultDirichletBCs()`. The final algebraic system is solved by the DefUtils function `DefaultSolve()`:

```
REAL(KIND=dp) :: Norm

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

It remains to implement the subroutine `ComputeLocal()` that makes the local computations. We will `contain` this subroutine in the main subroutine to simplify things:

```
SUBROUTINE MySolver(Model, Solver, dt, Transient)
...

CONTAINS

SUBROUTINE ComputeLocal(Element, N, Matrix, Vector)
  TYPE(Element_t), POINTER :: Element
  INTEGER :: N
  REAL(KIND=dp) :: Matrix(:,:)
  REAL(KIND=dp) :: Vector(:)
  ...
END SUBROUTINE ComputeLocal

END SUBROUTINE MySolver
```

The first thing to do in `ComputeLocal()` is to clear the matrix and vector:

```
Matrix = 0.0d0
Vector = 0.0d0
```

Next, we will get information about the node points:

```
TYPE(Nodes_t) :: Nodes
SAVE Nodes

Matrix = 0.0d0
Vector = 0.0d0

CALL GetElementNodes(Nodes, Element)
```

The Gauss points for our element are obtained by calling `GaussPoints()`

```
TYPE(GaussIntegrationPoints_t) :: IP

IP = GaussPoints(Element)
```

The local matrix and vector are integrated numerically by implementing a loop over the Gauss points, by evaluating the nodal basis functions in these points, and by computing the inner products:

```
    INTEGER :: i
    REAL(KIND=dp) :: detJ, Basis(N), dBasisdx(N,3)
    LOGICAL :: stat

    DO i = 1, IP % n
       stat = ElementInfo(Element, Nodes, &
          IP % u(i), IP % v(i), IP % w(i), &
          detJ, Basis, dBasisdx)

    END DO
```

In this loop, we will finally compute the inner products of the basis and their gradients, multiply the result by the weight of the Gauss point, and by the determinant of the Jacobian matrix of the mapping from the reference element:

```
       Matrix(1:N, 1:N) = Matrix(1:N, 1:N) + &
            MATMUL(dBasisdx, TRANSPOSE(dBasisdx)) * IP % s(i) * detJ

       Vector(1:N) = Vector(1:N) + Basis * IP % s(i) * detJ
```

The implementation is now complete.

Let us finally test the method by creating a finite element mesh e.g. with ElmerGrid or ElmerGUI (1, 2, and 3d are all fine), and by using the following SIF:

```
Header
  Mesh DB "." "."
End

Simulation
  Simulation Type = Steady state
  Steady State Max Iterations = 1
  Post File = case.ep
End

Body 1
  Equation = 1
End
```

```
Equation 1
  Active Solvers(1) = 1
End

Solver 1
  Equation = "MyEquation"
  Procedure = "MyLibrary" "MySolver"
  Variable = -dofs 1 "MyScalar"
End

Boundary condition 1
  Target boundaries(1) = 1
  MyScalar = Real 0
End
```