Basic Programming with Elmer

Mikko Lyly

Spring 2010

1 User defined functions

1.1 Calling convention

User defined functions (udf) can be used to compute complicated material parameters, body forces, boundary conditions, etc. Udfs 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 that computes g

END FUNCTION MyFunction
```

1.2 Compilation

Udfs are compiled into shared objects (Unix-like systems) or into a dlls (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 the input sign of the Command Prompt in Windows):

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

```
> elmerf90 MyLibrary.f90
```

1.3 Using udfs

User defined functions are called automatically by ElmerSolver when needed. To fix ideas, suppose that we want to compute the value of a material parameter as a function of time. In this case, the user defined function is activated from the Solver Input File e.g. as follows:

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

The value of time will be passed to the function in `f`. The function is supposed to compute the value of the material parameter in node `n` and return it to ElmerSolver in `g`.

The type `Model_t` is declared and defined in the source file `DefUtils.f90`. The structure contains a pointer to the mesh as well as 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 temperature), we can fetch the nodal value of that function by using the generic DefUtils routines (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
```

1.4 Excersices

Create a moderately small model for heat conduction (e.g. with ElmerGUI) and write a user defined function that returns a constant heat conductivity. Print out the node index and nodal coordinates to see if the function is actually called by ElmerSolver.

Modify your function so that it returns the value of the spatially varying heat conductivity $k = 1 + x$.

Finally, implement the temperature dependent heat conductivity $k = 1 + |T(x)|$ and visualize the result.

2 User defined solvers

2.1 Calling convention

All user defined subroutines that implement a custom solver are written 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.f90`.

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 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
```

2.4 Excercises

Create a temporary work directory containing the following mesh files:

```
$ less 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
```

```
$ less 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

$ less 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

$ less mesh.header
6 4 6
2
202 6
303 4
```

Then consider the following minimalistic Solver Input File:

```
$ less case.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
```

Finally, make sure that your work directory contains the following info file:

```
$ less ELMERSOLVER_STARTINFO
case.sif
1
```

Write and compile a user defined subroutine that simply prints out "Hello from My-Solver!" when called by ElmerSolver:

```
$ ElmerSolver
ELMER SOLVER (v 5.5.0) STARTED AT: 2010/05/24 10:17:10
MAIN:
MAIN: ==========================================
MAIN:  E L M E R  S O L V E R  S T A R T I N G
MAIN:  Library version: 5.5.0 (Rev: 4455)
MAIN: ==========================================
MAIN:
...
Hello from MySolver!
...
WriteToPost: Saving results in ElmerPost format to file ./case.ep
ElmerSolver: *** Elmer Solver: ALL DONE ***
ElmerSolver: The end
SOLVER TOTAL TIME(CPU,REAL):         0.11        0.38
ELMER SOLVER FINISHED AT: 2010/05/24 10:17:10
```

3 Reading constant data from SIF

Relevant functions and subroutines (defined in DefUtils.f90):

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

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

3.1 Reading constant scalars

Solver Input File:

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

Code (ElmerProgramming/case1/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

Solver Input File:

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

Code (ElmerProgramming/case2/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
End
```

Code (ElmerProgramming/case3/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
```

3.4 Excercises

Modify case1 such that your user defined subroutine reads and prints out an integer from the Solver block of your SIF (see `GetInteger()` in `DefUtils.f90`). Implement appropriate error handling.

4 Reading field data from SIF

Relevant functions and subroutines (defined in DefUtils.f90):

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

4.1 Reading scalar fields

Solver Input File:

```
Material 1
  MyParameter = Real 123.456
End
```

Code (ElmerProgramming/case4/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

  TYPE(Mesh_t), POINTER :: Mesh
  TYPE(Element_t), POINTER :: Element
  TYPE(ValueList_t), POINTER :: Material
  REAL(KIND=dp), ALLOCATABLE :: MyParameter(:)
  LOGICAL :: AllocationsDone = .FALSE.
  LOGICAL :: Found
  INTEGER :: N
  SAVE MyParameter, AllocationsDone

  IF(.NOT.AllocationsDone) THEN
     Mesh => GetMesh(Solver)
     N = Mesh % MaxElementNodes
     ALLOCATE(MyParameter(N))
     AllocationsDone = .TRUE.
  END IF

  N = GetNofActive(Solver)
  IF(N < 1) CALL Fatal("MySolver", "No elements in the mesh")

  Element => GetActiveElement(3)
  N = GetElementNofNodes(Element)

  Material => GetMaterial(Element)
  IF(.NOT.ASSOCIATED(Material)) CALL Fatal("MySolver", "No material block")

  MyParameter(1:N) = GetReal(Material, "MyParameter", Found)
  IF(.NOT.Found) CALL Fatal("MySolver", "MyParameter not found")
  PRINT *, "Element 3:"
  PRINT *, "Node indices:", Element % NodeIndexes(1:N)
  PRINT *, "Nodal values of MyParameter:", MyParameter(1:N)

END SUBROUTINE MySolver
```

Output:

```
 Element 3:
 Node indices:           1           4           5
 Nodal values of MyParameter:   123.45600000        123.45600000        123.45600000
```

4.2 Excercises

You can access your global solution vector in your subroutine as follows:

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

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
```

The vector `MyPermutation` is related to band width optimization and it is always on by default. You can turn the optimization off by adding the line `Bandwidth optimization = FALSE` in the Solver-block of your SIF. In this case the permutation vector `MyPermutation` becomes the identity map.

Write a user defined subroutine that loops over the elements, reads scalar field data from the Body Force-block of the SIF, and copies the nodal data into the global solution vector (that is, "solve" the equation $u = f$). Use the following Body Force block:

```
Body Force 1
  MyForce = Variable Coordinate 1
    Real
      -1.0 0.0
      1.0  123.456
    End
End
```

Visualize the solution with ElmerPost. The solution should grow linearly from left to right.

5 Partial Differential Equations

5.1 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

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

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

5.2 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 $\{\phi_1, \phi_2, \dots, \phi_n\}$. 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

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

The solution satisfies

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

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:

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

where

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

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.

5.3 Implementation

Let us next implement the method in Elmer by writing 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()` which performs 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 local 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 returned by the function `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
```

5.4 Excersices

Modify the above solver for the heat equation

$$-\nabla(k\nabla u) = f \quad \text{in } \Omega,$$

where $k > 0$ is the heat conduntion coefficient.

Modify the solver for a diffusion-reaction equation

$$-\nabla(k\nabla u) + au = f \quad \text{in } \Omega,$$

where $a \geq 0$ is the reaction coeficient. When a is large, there will be a sharp boundary layer. Adjust the mesh with appropriate refinements to resolve the layer accurately.