Elmer Programmer's Tutorial

Mikko Lyly, Mika Malinen and Peter Råback

CSC – IT Center for Science

2010–2015

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 8.1. Latest documentations and program versions of Elmer are available (or links are provided) at `http://www.csc.fi/elmer`.

Copyright information

The original copyright of this document belongs to CSC – IT Center for Science, Finland, 1995–2015. This document is licensed under the Creative Commons Attribution-No Derivative Works 3.0 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nd/3.0/`.

Elmer program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. Elmer software is distributed in the hope that it will be useful, but without any warranty. See the GNU General Public License for more details.

Elmer includes a number of libraries licensed also under free licensing schemes compatible with the GPL license. For their details see the copyright notices in the source files.

All information and specifications given in this document have been carefully prepared by the best efforts of CSC, and are believed to be true and accurate as of time writing. CSC assumes no responsibility or liability on any errors or inaccuracies in Elmer software or documentation. CSC reserves the right to modify Elmer software and documentation without notice.

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-subroutines (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)
```


CC BY-ND

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
```


CC BY-ND

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
```


CC BY-ND

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 (ListEx1.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
```


CC BY-ND

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 (ListEx2.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 (ListEx3.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
```


CC BY-ND

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.

CC BY-ND

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

The default variable for a normal solver will be created when it is decleared with a `Variable` statement in the SIF file.

In the command file a additional variables 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
```

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

CC BY-ND

Chapter 5

Standard IO and error handling

In this chapter the ways how standard information is printed on the console and errors are handled using `Info, Warning` and `Fatal` ssubroutines.

5.1 Info – Writing information on console

For writing information on console one could basically use the normal `PRINT` commands. However, in parallel this easily becomes complicated because often we want the information to be passed only by one process. Also it is easier to control the flow of information when it is hidden under a well defined function.

The basic information is passed with the following way:

```
CALL Info('MySolver','Starting the solver')
```

This has a default output level of 5 which. If the `Max Output Level` given in the `Simulation` section is smaller than this, no output will be printed. One can control the output level with an optional argument, `Level` as shown below

```
CALL Info('MySolver','Starting the solver',Level=4)
```

Often the output string should include also some numerical information. For this aim one can use a temporal global string `Message`, for example.

```
WRITE ( Message,'(A,ES12.3)') 'Scaling with factor: ',Coeff
CALL Info('MySolver',Message)
```

5.2 Error and Warning – Handling exceptions

The `Error` and `Warning` subroutines are basically used as `Info`. However, `Error` resulst to a termination of the program while `Warning` writes additional warnings on the console. Both also have a small output level of three.

```
IF( .NOT. ASSOCIATED(Solver % Variable) ) THEN
  CALL Fatal('MySolver','No variable associated for the solver!')
END IF
```


CC BY-ND

Chapter 6

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.

6.1 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
```


CC BY-ND

Chapter 7

Partial Differential Equations

In this chapter, we shall consider how the utilities of the Elmer solver are usually applied in order to create the description of a discrete PDE model. To this end, we shall introduce a model problem, describe the standard procedure for creating the computational version of the model, and finally consider the actual implementation by using the Elmer utilities.

The code for the problem is available as `ModelPDE.src`. Note that that solver has also reaction term implemented.

7.1 Model problem

As an example, consider solving a field $u = u(x,t)$ on $\Omega \times [0,T]$ that satisfy the convection-diffusion equation

$$\rho \frac{\partial u}{\partial t} + \kappa(\vec{a} \cdot \nabla)u - \mu \Delta u = f \quad \text{on } \Omega \times [0,T], \tag{7.1}$$

the initial condition

$$u(x,0) = u_0(x)$$

for every $x \in \Omega$, and the boundary conditions

$$u = 0 \quad \text{on } \Gamma_D \times [0,T] \tag{7.2}$$

and

$$\mu \frac{\partial u}{\partial n} = \alpha(g - u) + q \quad \text{on } \Gamma_N \times [0,T], \tag{7.3}$$

with $\Gamma_D \cup \Gamma_N$ giving the boundary of Ω. The problem specification therefore involves giving the body $\Omega \subset R^d$ (with $d \in \{1,2,3\}$) and the final time T, the initial state u_0, the source data f, g, q and α, material parameters ρ, κ and μ, and the vector field $\vec{a}$.

This example case is thus

- evolutionary via the presence of the time derivative term
- parameter dependent via scalars $\rho : \Omega \times \mathbb{R} \to \mathbb{R}$, $\mu : \Omega \times \mathbb{R} \to \mathbb{R}$ and a vector $\vec{a} : \Omega \times \mathbb{R} \to \mathbb{R}^d$

Taken that the vector field $\vec{a}$ may describe a solution to another PDE model, our treatment given in the following can also be considered to imitate the solver development when an interaction between different PDE models is taken into account by employing the standard segregated solution strategy of Elmer (cf. the Chapter 1 of ElmerSolver Manual).

7.2 Obtaining a discrete version of the model

In the following, we describe basic steps for obtaining the discrete version of a PDE model which can then be implemented by making use of a collection of standard Elmer utilities.

BY-ND

Step I: Semi-discretization in time. Implicit time discretization is usually applied in Elmer. For example, in the case of the backward Euler method, we start by replacing (7.1) by the time-discretized version

$$\rho\frac{u^{n+1}}{\Delta t} + \kappa(\vec{a}\cdot\nabla)u^{n+1} - \mu\Delta u^{n+1} = f^{n+1} + \rho\frac{u^n}{\Delta t}, \tag{7.4}$$

where Δt is the time step size for advancing from time $t = t^n$ to $t^{n+1} = t^n + \Delta t$.

Step II: Weak formulation. To obtain a version of the semi-discrete problem which is suitable for the spatial discretization using finite elements, the weak formulation of the semi-discrete problem is first written. In the case of the example case (7.4) we are lead to seeking a sufficiently smooth $u^{n+1} \in X$ such that

$$\begin{aligned}\int_\Omega \rho\frac{u^{n+1}}{\Delta t}v\,d\Omega + \int_\Omega \kappa(\vec{a}\cdot\nabla)u^{n+1}v\,d\Omega + \int_\Omega \mu\nabla u^{n+1}\cdot\nabla v\,d\Omega + \int_{\Gamma_N}\alpha u^{n+1}v\,dS \\ = \int_\Omega f^{n+1}v\,d\Omega + \int_\Omega \rho\frac{u^n}{\Delta t}v\,d\Omega + \int_{\Gamma_N}\alpha g v\,dS + \int_{\Gamma_N} q v\,dS\end{aligned} \tag{7.5}$$

for any $v \in X$. The right choice of the solution space X generally depends on the PDE model considered. Here we take $X \subset H^1(\Omega)$ so that X contains square-integrable functions over Ω whose all first partial derivatives also are square-integrable. In addition, any $u \in X$ is required to satisfy the constraint (7.2).

Step III: Finite element approximation. To obtain the spatial discretization via applying the Galerkin FE approximation, we divide Ω into finite elements and introduce a set of mesh dependent finite element basis functions ϕ_j such that $X_h = \mathrm{span}\{\phi_1, \phi_2, \ldots, \phi_N\} \subset X$. The approximate solution is then sought from the space X_h as a linear combination of the basis functions and determined from a finite-dimensional version of the weak formulation. In the case of the example problem (7.5) we therefore seek

$$u_h^{n+1} = \sum_{i=1}^N u_i^{n+1}\phi_i \qquad (u_i^{n+1} \in \mathbb{R})$$

such that

$$\begin{aligned}\int_\Omega \rho_h\frac{u_h^{n+1}}{\Delta t}v_h\,d\Omega + \int_\Omega \kappa_h(\vec{a}_h\cdot\nabla)u_h^{n+1}v_h\,d\Omega + \int_\Omega \mu_h\nabla u_h^{n+1}\cdot\nabla v_h\,d\Omega + \int_{\Gamma_N}\alpha_h u_h^{n+1}v_h\,dS \\ = \int_\Omega f_h^{n+1}v_h\,d\Omega + \int_\Omega \rho_h\frac{u^n}{\Delta t}v_h\,d\Omega + \int_{\Gamma_N}\alpha_h g_h v_h\,dS + \int_{\Gamma_N} q_h v_h\,dS\end{aligned} \tag{7.6}$$

for any $v_h \in X_h$. The use of a subscript h in conjunction with the input data indicates that typically finite element interpolation is also employed in order to approximate the input data.

Step IV: Linearization. Generally the fully discrete problem resulting from the time and spatial discretization leads to solving a nonlinear system of algebraic equations. In such case, a nonlinear iteration is usually employed in order to obtain a solution to the nonlinear problem. The above treatment has nevertheless avoided the discussion of nonlinearities. In the current version only fixed value iteration is possible.

We conclude that the fully discretized (and potentially linearized) version of the PDE model finally leads us to solving linear algebra problems

$$(\frac{1}{\Delta t}M + K)U^{n+1} = F + \frac{1}{\Delta t}MU^n \tag{7.7}$$

where the solution vector U^{n+1} contains the coefficients in the finite element expansion of the solution, while M and $K = K$ are referred to as the scaled mass matrix (with ρ acting as a scaling factor) and the stationary

CC BY-ND

stiffness matrix, respectively. The entries of these matrices and the stationary part F of the right-hand side vector are computed as

$$\begin{aligned} M_{ij} &= \int_\Omega \rho_h \phi_j \phi_i \, d\Omega \\ K_{ij} &= \int_\Omega \kappa_h (\vec{a}_h \cdot \nabla) \phi_j \phi_i \, d\Omega + \int_\Omega \mu_h \nabla \phi_j \cdot \nabla \phi_i \, d\Omega + \int_{\Gamma_N} \alpha_h \phi_j \phi_i \, dS \\ F_i &= \int_\Omega f_h^{n+1} \phi_i \, d\Omega + \int_{\Gamma_N} \alpha_h g_h \phi_i \, dS + \int_{\Gamma_N} q_h \phi_i \, dS. \end{aligned} \tag{7.8}$$

7.3 Implementation

In practice, the entries of the linear system resulting from the finite element approximation are computed by calculating contributions elementwise and assembling the resulting element contributions into the actual global linear system. In Elmer the book-keeping which is required in the implementation of this strategy so that the elementwise contributions are added into the correct locations of the global system is basically automated. The treatment of contributions relating to the presence of time derivatives is standardized similarly. Thus, the only low-level instructions relating to the implementation of a PDE solver are typically associated with generating the element-level versions of the (scaled) mass matrix, the stationary stiffness matrix and the stationary part of the right-hand side vector.

If ψ_i are element basis functions such that they coincide with the non-trivial restrictions of the global basis functions on the element E considered, the creation of the example system (7.6) requires that a routine for generating the element-level versions of (7.8) defined as

$$\begin{aligned} M_{ij}^E &= \int_E \rho_h \psi_j \psi_i \, d\Omega, \\ K_{ij}^E &= \int_E \kappa_h (\vec{a}_h \cdot \nabla) \psi_j \psi_i \, d\Omega + \int_E \mu_h \nabla \psi_j \cdot \nabla \psi_i \, d\Omega + \int_{\Gamma_N} \alpha_h \psi_j \psi_i \, dS, \\ F_i^E &= \int_E f_h^{n+1} \psi_i \, d\Omega + \int_{\Gamma_N} \alpha_h g_h \psi_i \, dS + \int_{\Gamma_N} q_h \psi_i \, dS \end{aligned} \tag{7.9}$$

is written. The integrals over the elements are evaluated by integrating over a fixed reference element $\hat{E}$. For example, given an element mapping $f_E : \hat{E} \to E$, the element mass matrix is computed as

$$M_{ij}^E = \int_{\hat{E}} \rho_h(f_E(\hat{x})) \; \psi_i(f_E(\hat{x})) \; \psi_j(f_E(\hat{x})) \; |J_E(\hat{x})| \, d\hat{\Omega}$$

where $|J_E|$ is the determinant of the Jacobian matrix of f_E. In most cases, f_E is either an affine or isoparametric map from the reference triangle, square, tetrahedron, hexahedron, etc., into the actual element. Finally, the integral over the reference element is computed numerically with an appropriate quadrature, so that

$$M_{ij}^E = \sum_{k=1}^{N_G} w_k \; \rho_h(f_E(\hat{x}_k)) \; \psi_i(f_E(\hat{x}_k)) \; \psi_j(f_E(\hat{x}_k)) \; |J_E(\hat{x}_k)|$$

where $\hat{x}_k$ are the integration points and w_k are the integration weights. Elmer uses the Gauss quadrature by default.

To sum up, a PDE solver can be implemented by creating a separate software module which contains the assembly loop over the elements and, if the problem is nonlinear, instructions for performing the nonlinear iteration. In addition, it usually contains certain standard high-level subroutine calls for performing such tasks as setting boundary conditions and solving the linear systems assembled.

CC BY-ND

If the consideration of the nonlinear iteration loop is now omitted for the simplicity of presentation, in the case of our model PDE problem the body of the solver code, which is encapsulated into a subroutine having a standard set of arguments and now given the name `AdvDiffSolver`, may simply consist of the following instructions:

```
!------------------------------------------------------------------------------
SUBROUTINE AdvDiffSolver( Model,Solver,dt,TransientSimulation )
!------------------------------------------------------------------------------
  USE DefUtils
  IMPLICIT NONE
!------------------------------------------------------------------------------
  TYPE(Solver_t) :: Solver
  TYPE(Model_t) :: Model
  REAL(KIND=dp) :: dt
  LOGICAL :: TransientSimulation
!------------------------------------------------------------------------------
! Local variables
!------------------------------------------------------------------------------
  ...
!------------------------------------------------------------------------------
   ElementCount = GetNOFActive()      ! Obtain the count of volume elements
   CALL DefaultInitialize()           ! Perform standard initialization

   DO t=1,ElementCount
      Element => GetActiveElement(t)  ! Gives a pointer to the element processed
      n  = GetElementNOFNodes()       ! The count of Lagrange basis functions
      nd = GetElementNOFDOFs()        ! The count of actual basis functions

      CALL IntegrateAndAssemble(Element, n, nd)
   END DO

   CALL DefaultFinishBulkAssembly()   ! To finalize assembly process over a volume

   ElementCount = GetNOFBoundaryElements()   ! Obtain the count of elements decribing the boundary
   DO t=1,ElementCount
      Element => GetBoundaryElement(t)       ! Gives a pointer to the element processed
      IF(ActiveBoundaryElement()) THEN
         n  = GetElementNOFNodes()
         nd = GetElementNOFDOFs()
         CALL IntegrateAndAssembleOverBoundary(Element, n, nd)
      END IF
   END DO
   CALL DefaultFinishBoundaryAssembly()  ! To finalize assembly process over the boundary

   CALL DefaultFinishAssembly()       ! To finalize the entire assembly process
   CALL DefaultDirichletBCs()         ! Set Dirichlet boundary conditions
   Norm = DefaultSolve()              ! Solve the linear system assembled

CONTAINS
  !------------------------------------------------------------------------------
  SUBROUTINE IntegrateAndAssemble(Element, n, nd)
  !------------------------------------------------------------------------------
  ...
  END SUBROUTINE IntegrateAndAssemble

  !------------------------------------------------------------------------------
  SUBROUTINE IntegrateAndAssembleOverBoundary(Element, n, nd)
  !------------------------------------------------------------------------------
  ...
  END SUBROUTINE IntegrateAndAssembleOverBoundary


!------------------------------------------------------------------------------
END SUBROUTINE AdvDiffSolver
!------------------------------------------------------------------------------
```

It should be noted that the specification of the element matrices and vectors and performing their assembly is here encapsulated into the contained subroutines which remain to be detailed. Otherwise we employ high-level utilities that are usually applied similarly regardless of details which are specific to a PDE model.

Considering the implementation of the subroutine `IntegrateAndAssemble(Element,n,nd)`, we first introduce the following local variables

```
    REAL(KIND=dp) :: MASS(nd,nd), STIFF(nd,nd), FORCE(nd)
    REAL(KIND=dp) :: Basis(nd), dBasisdx(nd,3), DetJ, Weight

    INTEGER :: dim

    LOGICAL :: Stat, Found
```


CC BY-ND

```
TYPE(GaussIntegrationPoints_t) :: IP

TYPE(ValueList_t), POINTER :: BodyForce

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

which are useful for implementing nearly any solver. We note that the argument `nd` provides the right parameter for specifying the sizes of the element mass matrix `MASS` and stationary stiffness matrix `STIFF` as well as the associated right-hand side vector `FORCE`. On the other hand, the variables `Nodes` and `IP` may then be used in the subroutine body to obtain information about element nodes and the Gauss quadrature:

```
CALL GetElementNodes(Nodes)
IP = GaussPoints(Element)
```

The values of the (H^1-regular) basis functions and their derivatives with respect to the global coordinates x_j as well as the determinant of the Jacobian matrix in an integration point may also be obtained by creating a loop over the Gauss points as

```
INTEGER :: t

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

In order to create the finite element expansion of an input parameter or the source data, we need to create a vector of the associated scalar coefficients prior to entering into the loop over the integration points. It should be noted that the classic Lagrange interpolation basis functions λ_i, $i = 1, \ldots, n$, are applied in this connection, with the available number of these basis functions defined by the program variable `n` above. In cases of discretizations based on the H^1-regular basis functions we generally have $\lambda_i = \psi_i$, while ψ_i, $n < i \leq nd$ may be additional basis functions which are not derived from the classic Lagrange interpolation property (they may be hierarchic basis functions corresponding to the p-version of the finite element method). For example, to write the expansions

$$f_h = \sum_i^n f_i \lambda_i, \quad \rho_h = \sum_i^n \rho_i \lambda_i$$

corresponding to the source data f and the material parameter ρ which are assumed to be referred to as the `'Source'` and `'Density'` in the solver input file, we may create the vectors `LOAD` and `TimeDerivativePar` which contain the coefficients f_i and ρ_i as

```
REAL(KIND=dp) :: LOAD(n), TimeDerivativePar(n)

TimeDerivativePar(1:n) = GetReal(GetMaterial(), 'Density', Found)

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

The values of f_h and ρ_h at an integration point are then obtained in the quadrature loop as

```
  LoadAtIP = SUM( Basis(1:n) * LOAD(1:n) )
  rho = SUM( Basis(1:n)*TimeDerivativePar(1:n) )
```

If we assume that the vector `DiffusionPar` contains the current coefficients for approximating the diffusion parameter μ, a complete set of instructions for integrating the scaled mass matrix, the part of the stiffness matrix corresponding to the diffusion term and the part of the element right-hand side vector corresponding to the body source finally reads

```
!------------------------------------------------------------------------------
SUBROUTINE IntegrateAndAssemble(Element, n, nd)
!------------------------------------------------------------------------------
  INTEGER :: n, nd
  TYPE(Element_t), POINTER :: Element
!------------------------------------------------------------------------------
  REAL(KIND=dp) :: MASS(nd,nd), STIFF(nd,nd), FORCE(nd)
  ...
!------------------------------------------------------------------------------
  CALL GetElementNodes(Nodes)
  IP = GaussPoints(Element)
```


CC BY-ND

```
    TimeDerivativePar(1:n) = GetReal(GetMaterial(), 'Density', Found)
    BodyForce => GetBodyForce()
    IF ( ASSOCIATED(BodyForce) ) &
       Load(1:n) = GetReal( BodyForce, 'Source', Found )

    dim = CoordinateSystemDimension()
    MASS  = 0._dp
    STIFF = 0._dp
    FORCE = 0._dp

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

      LoadAtIP = SUM( Basis(1:n) * LOAD(1:n) )
      rho = SUM( Basis(1:n)*TimeDerivativePar(1:n) )
      D = SUM( Basis(1:n)*DiffusionPar(1:n) )

      Weight = IP % s(t) * DetJ

      STIFF(1:nd,1:nd) = STIFF(1:nd,1:nd) + Weight * &
             D * MATMUL( dBasisdx, TRANSPOSE( dBasisdx ) )

      DO p=1,nd
        DO q=1,nd
          MASS(p,q) = MASS(p,q) + Weight * rho * Basis(q) * Basis(p)
        END DO
      END DO

      FORCE(1:nd) = FORCE(1:nd) + Weight * LoadAtIP * Basis(1:nd)
    END DO

    IF(TransientSimulation) CALL Default1stOrderTime(MASS,STIFF,FORCE)
    CALL DefaultUpdateEquations(STIFF,FORCE)

  END SUBROUTINE IntegrateAndAssemble
```

Here the general subroutines `Default1stOrderTime` and `DefaultUpdateEquations` are used after the numerical integration to first include the effect of the first-order time derivative and then to assemble the element contributions to the global linear system. Now, only a small modification would be needed in order to implement the convective term.

The subroutine `IntegrateAndAssembleOverBoundary(Element, n, nd)` can be written in a similar way. To obtain a ready code, see the directory `../trunk/fem/tests/ModelPDE` contained in the source code repository of Elmer.

CC BY-ND