3.2 Overview of the software *OXFEMM*

3.2.1 Objective of the software

At first sight, *OXFEMM* is a basic Finite Element (FE) solver such as the commercial software Abaqus© with the difference that it will be open source. In the same way as commercial software, the user defines a body, a material, a mesh, boundary conditions, a time simulation and the numerical solver. The weak form of the mechanical balance is solved for each time step following Equation (1):

$$\int_{\partial\Omega_0} \overline{\boldsymbol{T}} \cdot \boldsymbol{\eta}\, \mathrm{d}S + \int_{\Omega_0} \rho_0 \boldsymbol{b} \cdot \boldsymbol{\eta}\, \mathrm{d}V = \int_{\Omega_0} \boldsymbol{P} : \mathrm{Grad}\, \boldsymbol{\eta}\, \mathrm{d}V + \int_{\Omega_0} \rho_0 \ddot{\boldsymbol{x}} \cdot \boldsymbol{\eta}\, \mathrm{d}V \tag{1}$$

for all admissible virtual displacement $\boldsymbol{\eta}$, with $\boldsymbol{\eta}(\boldsymbol{X}) = \boldsymbol{0}$ for all $\boldsymbol{X} \in \partial\Omega_d$. All of the variables are detailed in the Nomenclature section at the end of this report. In essence, this equation describes inertia terms, external forces and internal forces. This equation is described in more detail in Appendix A.

Since this balance is written in integral form, the Finite Element Method is used to spatially discretise the domain into elements and to obtain an equation where integrals are evaluated over each element rather than the entire volume, see figure 1. *OXFEMM* allows FE and MM[6] calculations. The power of the software lies in its ability to perform calculations combining the two methods of resolution (FEMM). A recent option makes it possible to divide a single domain into several parts or subdomains, and to use Meshless Method when it comes to large deformations where the resolution of the finite element problem is not stable. Thus, we can have a single domain but resolved in several different ways, depending on the need and complexity of the problem.

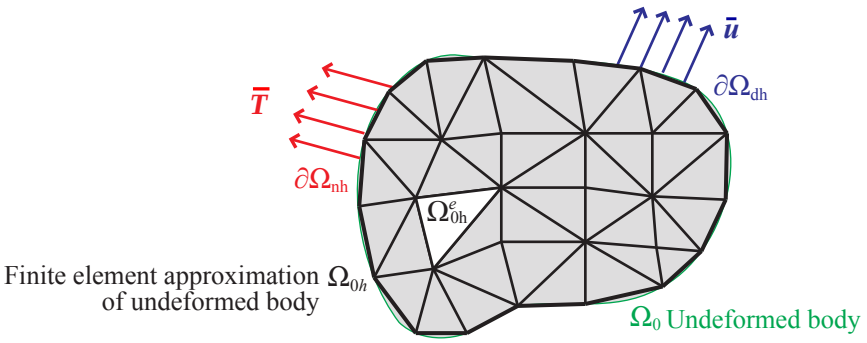


Figure 1: Finite element discretisation of the undeformed body Ω_0 into elements Ω^e_{0h}, such that $\Omega_{0h} = \bigcup_e \Omega^e_{0h}$.

Either FEM or MM is used, the numerical integration is made over the Gauss Points to refrain from using elements in MM. Equation (1) becomes Equation (2), a numerical equation written at each node that can be used in the algorithm to compute displacements. Since the aim is to reach the balance (2) at each timestep, the problem is to find the vector of displacements $\boldsymbol{U}$ containing the displacement $\boldsymbol{u}^a$ of each node a such that (2) is verified at each node a. Three solvers to numerically solve this equation have been implemented on *OXFEMM*: an implicit static, an implicit dynamic and an explicit one.

$$\boldsymbol{M} \cdot \ddot{\boldsymbol{x}} + \boldsymbol{f}^{\mathrm{int}}(\boldsymbol{x}) = \boldsymbol{f}^{\mathrm{ext}} \tag{2}$$

[6]Meshless Methods

3.2.2 How to use *OXFEMM*

For the time being, the software doesn't have any graphical user interface (GUI), computations are only run from a terminal on a Linux machine. In this terminal, the user specifies which input file he/she wants to simulate and how many processors he/she wants to use if he/she had previously chosen to compile the software in parallel computation (sequential computations are chosen by default). When the simulation is finished, a message invites the user to open "outputs" folder to get the results written in VTK format. The user has to execute another software package to visualise the results since *OXFEMM* is only for solving computations. ParaView is a free software package that allows one to open VTK files. The user can find on *OXFEMM* Manual how to use this software, how to display the nodes, elements, scalar and vector fields etc.

The input files (.inp) run by *OXFEMM* are stored in an "input" folder and follow a specific format described in the manual. It follows Abaqus© format for the declaration of the nodes, elements and the sets of nodes, but the rest is different. The distribution of FEM and/or MM elements is then specified just before the declaration of each solver used in the study. Number of outputs, simulation time and boundary conditions are enumerated for each solver. Since the whole file needs to be rigorously written, a mistake is easily made especially if there are more than 1000 nodes. In this respect, the input file is generated in two steps. First, Abaqus© is used only to create the mesh and the set of nodes inside an input file *Job.inp* – in the future, *OXFEMM* will have to be able to use an open source meshing solver. Second, a Python script is executed to read *Job.inp* file and to create *example.inp*. The input file then can be read by *OXFEMM*. This script displays an interface where the user chooses which Abaqus© mesh file he/she wants to upload. After that, all parameters needed can be chosen with drop-down menus. Figure 2 sums up all of the steps to follow in order to make a simulation with *OXFEMM*.

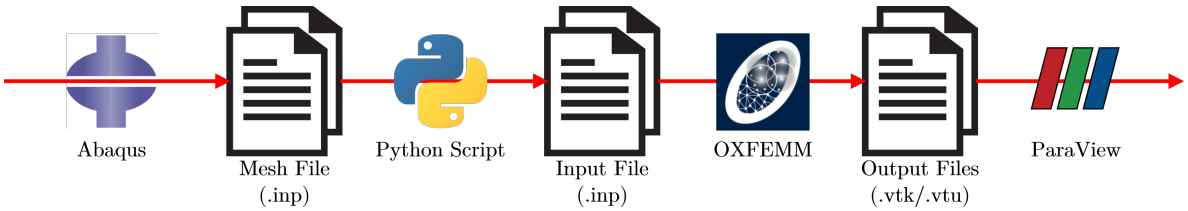


Figure 2: Blueprint to use *OXFEMM*

4 Main assets of *OXFEMM*

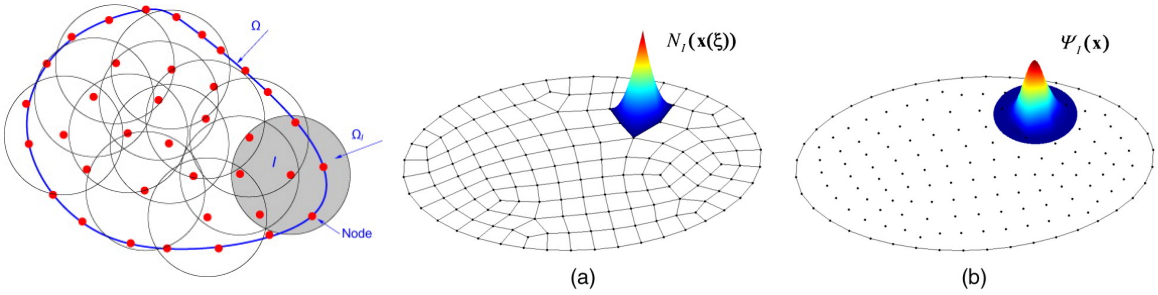
4.1 Meshless Method

The domain Ω_0 can be discretised by means of FEM, MM or both (Coupling). It is possible to have two different domains, one for FEM and one for MM, there is then a group of nodes that are shared by the FEM and MM domains and appear in both lists of nodes of each domain. The values of the unknown variables at the shared nodes will be equal: the coupling between both regions is then naturally obtained.

Regarding the method used (FEM/MM/Coupling), the initial problem is defined as follows:

$$\boldsymbol{u}^h(\boldsymbol{x}) = \sum_{a=1}^{n} N_a(\boldsymbol{x})\boldsymbol{u}_a \tag{3}$$

where $\boldsymbol{u}^h(\boldsymbol{x})$ is the approximation of the real displacement $\boldsymbol{u}$, with $\boldsymbol{u}_a$ the displacements at nodes. $\boldsymbol{x}$ is the position of the point at which these quantities are being evaluated –in our case a Gauss Point– and n is the number of points in the neighbourhood of $\boldsymbol{x}$ (in other words, n is the number of nodes for which the evaluating point $\boldsymbol{x}$ is inside their domains of influence). For FEM, n is the number of nodes of the element to which the evaluating point belongs. The shape function $N(\boldsymbol{x})$ will depend again on the method used. This is one of the main differences between *OXFEMM* and other FEM programs: the whole program has been built around the Gauss points data structures and not around elements. Figure 3 shows a MM spatial discretisation and an example of an MM shape function.



(a) Meshless approximation of undeformed body, taken from [9]

(b) Shape function associated to an integration point for FEM (a) and MM (b), taken from [11]

Figure 3: Meshless Method

4.2 Solvers available

OXFEMM is mainly focused on non-linear problems for large deformation scenarios. Consequently, the program implements several algorithms and solvers to encompass such problems: an explicit, an implicit static and an implicit dynamic.

4.2.1 Implicit Dynamic

In order to solve Equation (2) the Newmark time integration scheme is defined as follows:

$$\boldsymbol{M} \cdot \ddot{\boldsymbol{x}}_{n+1} + \boldsymbol{f}^{int}(\boldsymbol{x}_{n+1}) = \boldsymbol{f}^{ext}_{n+1} \tag{4}$$

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta t \dot{\boldsymbol{x}}_n + \Delta t^2 \left(\left(\frac{1}{2} - \beta \right) \ddot{\boldsymbol{x}}_n + \beta \ddot{\boldsymbol{x}}_{n+1} \right) \tag{5}$$

$$\dot{\boldsymbol{x}}_{n+1} = \dot{\boldsymbol{x}}_n + \Delta t \left((1-\gamma) \ddot{\boldsymbol{x}}_n + \gamma \ddot{\boldsymbol{x}}_{n+1} \right) \tag{6}$$

where subscripts "n" and "$n+1$" indicate that the quantities are evaluated at time t_n and t_{n+1} respectively, where $\Delta t = t_{n+1} - t_n$ is the time step. Parameters $0 \leq \beta \leq 1/2$ and $0 \leq \gamma \leq 1$ set the characteristics of the Newmark scheme and by tuning them the stability and accuracy of the solution can be managed.

As $\boldsymbol{f}^{int}$ is often formulated as a function of the displacement vector $\boldsymbol{u} = \boldsymbol{x} - \boldsymbol{X}$ (e.g., $\boldsymbol{f}^{int} = \boldsymbol{K} \cdot \boldsymbol{u}$) and noting that $\dot{\boldsymbol{u}} = \dot{\boldsymbol{x}}$ and $\ddot{\boldsymbol{u}} = \ddot{\boldsymbol{x}}$, it is sometime more convenient to solve this system in $\boldsymbol{u}$ instead of $\boldsymbol{x}$.

Let $\boldsymbol{r}^k$ be the remainder (or residual) of the balance of linear momentum equation at iteration step k,

$$\boldsymbol{r}^k = \boldsymbol{M} \cdot \ddot{\boldsymbol{x}}^k_{n+1} + \boldsymbol{f}^{int}(\boldsymbol{x}^k_{n+1}) - \boldsymbol{f}^{ext}_{n+1} \tag{7}$$

Using Newton's method described in Appendix B, the updated nodal deformation can be computed as

$$\boldsymbol{x}^{k+1}_{n+1} = \boldsymbol{x}^k_{n+1} - [\boldsymbol{K}^{eq}(\boldsymbol{x}^k_{n+1})]^{-1} \cdot \boldsymbol{r}^k \tag{8}$$

where $\boldsymbol{K}^{eq}(\boldsymbol{x}^k_{n+1})$ is given by

$$\boldsymbol{K}^{eq}(\boldsymbol{x}^k_{n+1}) = \frac{1}{\beta \Delta t^2} \boldsymbol{M} + \boldsymbol{K}(\boldsymbol{x}^k_{n+1}) \tag{9}$$

where $\boldsymbol{K} = \frac{\partial \boldsymbol{f}^{int}}{\partial \boldsymbol{x}}$ is the stiffness matrix (see Appendix A for more details). The updated nodal accelerations and velocities can then be updated:

$$\ddot{\boldsymbol{x}}^{k+1}_{n+1} = \ddot{\boldsymbol{x}}^k_{n+1} + \frac{1}{\beta \Delta t^2} \Delta \boldsymbol{x}_{n+1} \quad \text{and} \quad \dot{\boldsymbol{x}}^{k+1}_{n+1} = \dot{\boldsymbol{x}}^k_{n+1} + \frac{\gamma}{\beta \Delta t} \Delta \boldsymbol{x}_{n+1} \tag{10}$$

If the norm of the residual is larger than a given tolerance ε_{tol}, another iteration can be run, otherwise the iterations can be stopped.

4.2.2 Implicit Static

As the accelerations can be neglected, Equation (1) can be directly written as follow:

$$\boldsymbol{f}^{int}(\boldsymbol{x}_{n+1}) = \boldsymbol{f}^{ext}_{n+1} \tag{11}$$

The method is exactly the same, the residual now given by $\boldsymbol{Res}^k = \boldsymbol{f}^{int}(\boldsymbol{x}^k_{n+1}) - \boldsymbol{f}^{ext}_{n+1}$ and the stiffness matrix $\boldsymbol{K}(\boldsymbol{x}^k_{n+1})$ at iteration step k.

4.2.3 Explicit

If $\beta = 0$, the generalised Newmark scheme becomes explicit. For any given $\boldsymbol{x}_n$, $\dot{\boldsymbol{x}}_n$, and $\ddot{\boldsymbol{x}}_n$ at time t_n, the solution of $\boldsymbol{x}_{n+1}$ is obtained from Equation (5):

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \Delta t \dot{\boldsymbol{x}}_n + \frac{\Delta t^2}{2} \ddot{\boldsymbol{x}}_n \tag{12}$$

with the variable $\ddot{\boldsymbol{x}}_n$ directly computed from the previous timestep as:

$$\ddot{\boldsymbol{x}}_n = \boldsymbol{M}^{-1} \cdot \left(\boldsymbol{f}_n^{ext} - \boldsymbol{f}^{int}(\boldsymbol{x}_n) \right) \tag{13}$$

In order to avoid inverting the mass matrix $\boldsymbol{M}$, the matrix can be lumped (i.e., diagonalized). Once the mass matrix has been lumped, Equation (13) can be solved node by node.

γ must be larger or equal to $1/2$ for a stable scheme. When $\gamma = 1/2$ is chosen, this scheme is second-order accurate and is called a central difference scheme. This scheme is conditionally stable, i.e., Δt needs to be smaller than a critical time step Δt_c:

$$\Delta t_c = \min_e \left(\frac{l_e}{C_e} \right) \tag{14}$$

where for each element e, l_e is its characteristic size and C_e is the velocity of sound in the material (provided by the constitutive law).

4.3 Parallel computation

By default, all of the computations are done sequentially, meaning that only one processor takes charge of all computations step by step. This is not a problem for small simulations, but when there is a large number of DOFs in the mesh ($\sim$ 10,000), the storage requirements and the time spent for computations can increase drastically. To solve this issue, parallel computation is used. This allows one to assign the storage of the data and the computations into different processors.

4.3.1 Spatial partitioning

In parallel computation, each processor only computes global vectors –such as the global stiffness matrix or the global force vectors– on a specific part of the body defined by a list of elements. Since one node can belong to different elements, some nodes are shared between different processors. At the end of the parallel computations, all processors gather their contribution to a single global object (vector or matrix). Since there are shared nodes, each processor add its own contribution to the global vector/matrix to build the correct solution. Figure 4 and table 1 show an example of parallel distribution made on 6 triangular elements of a 2D beam.

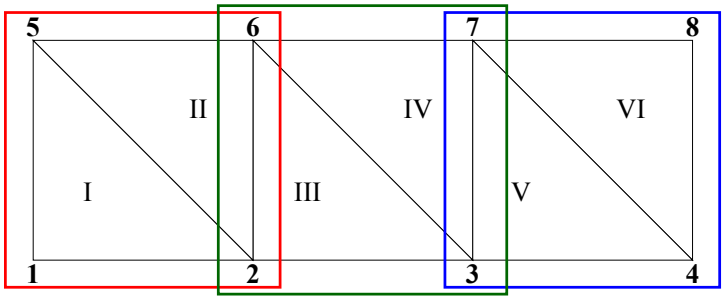


Figure 4: 2D example for parallel partitioning

Processor	1	2	3
Elements	[I,II]	[III,IV]	[V,VI]
Nodes	[1,2,5,6]	[2,3,6,7]	[3,4,7,8]
$\boldsymbol{U}$	$[\boldsymbol{u}_1,\boldsymbol{u}_2,\boldsymbol{u}_5,\boldsymbol{u}_6]$	$[\boldsymbol{u}_2,\boldsymbol{u}_3,\boldsymbol{u}_6,\boldsymbol{u}_7]$	$[\boldsymbol{u}_3,\boldsymbol{u}_4,\boldsymbol{u}_7,\boldsymbol{u}_8]$

Table 1: Spatial partitioning of Degrees of Freedom (DOFs)

4.3.2 Memory storage

There is a different partitioning for the memory storage. After gathering the contribution of each processor into a single object, this object is divided and stored in the processors, each containing almost the same number of values as others. Table 2 shows the memory storage partitioning of the same example shown on Figure 4. Since this is a 2D problem with 8 nodes, there are $2 \times 8 = 16$ DOFs[11]. Two processors will then store 5 values and the third one 6 values.

Processor	1	2	3
$\boldsymbol{U}$	$[[u_{1x},u_{2x},u_{3x},u_{4x},u_{5x}]$	$[u_{6x},u_{7x},u_{8x},u_{1y},u_{2y}]$	$[u_{3y},u_{4y},u_{5y},u_{6y},u_{7y},u_{8y}]]$

Table 2: Partitioning of memory storage

4.4 Structure of *OXFEMM*

OXFEMM is written in C++ with an Object-Oriented Programming (OOP) in several classes. All the files are classified as shown in Figure 5.

Figure 6 shows the general structure of *OXFEMM* which is similar to a conventional FEM software. The main difference lies in the calculation of the neighbourhood and shape functions of MM Gauss Points and nodes.

[11] Degree(s) Of Freedom

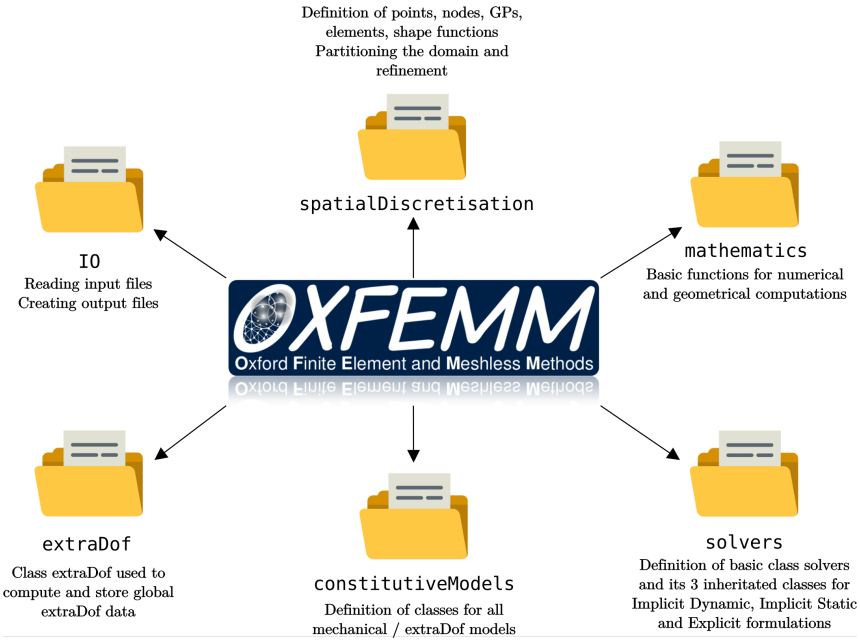


Figure 5: Organisation of the source code of the program

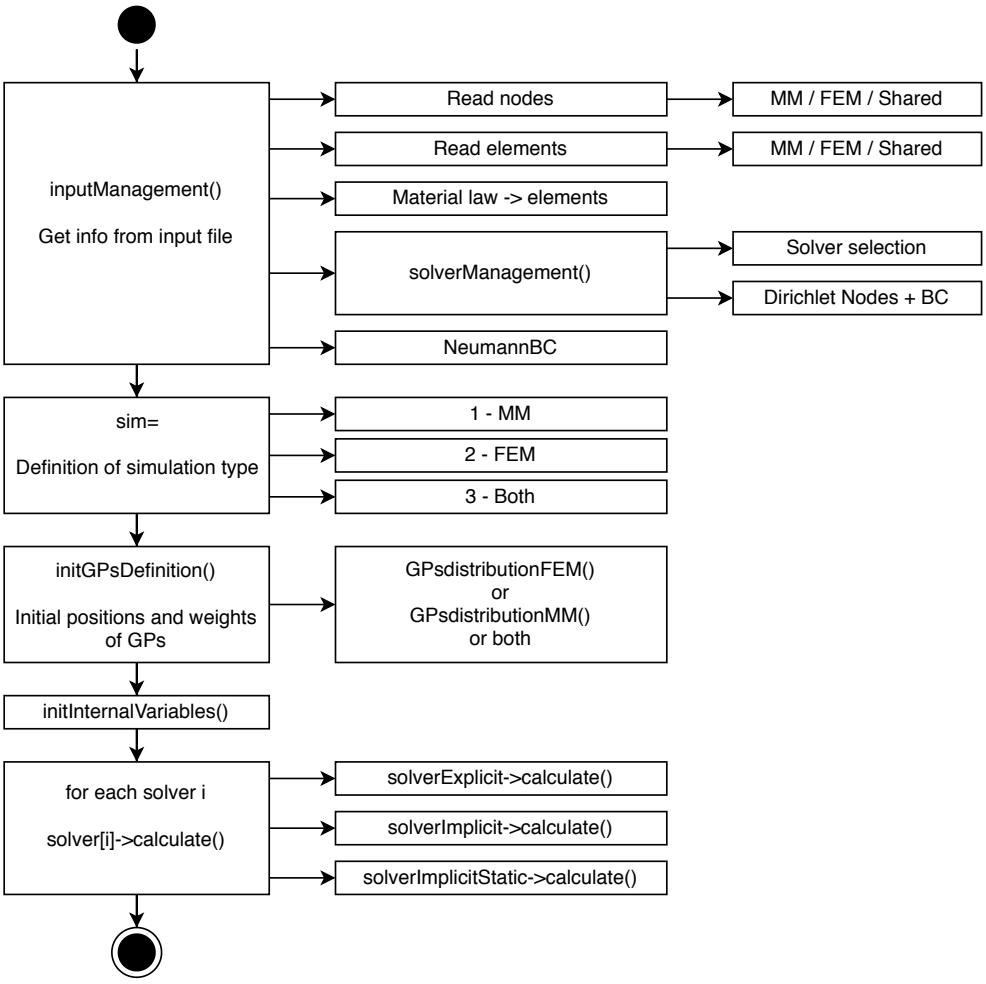


Figure 6: General structure of Oxfemm

Figures 25 to 27 in Appendix C show the structure for the explicit and implicit solvers (the implicit static solver is a minor transformation of the implicit solver). Again, they are similar to a conventional FEM software except for the use of nodes, GPs and shape functions due to the inclusion of MM part. In essence, these diagrams describe the following:

1. Computation of the Mass Matrix: Used for solvers Explicit and Implicit Dynamic.
2. Beginning of the Time loop.
3. Prediction: Different for each solver (values of previous timestep for Implicit Static).
4. Beginning of Newton-Raphson's loop: Only for Implicit solvers.
5. Update of the internal forces: Computation of $\boldsymbol{P}$, $\boldsymbol{F}$ and $\boldsymbol{F}^{\text{int}}$ (and also $\boldsymbol{K}$ for Implicit solvers).
6. Update of the external forces: Computation of $\boldsymbol{F}^{\text{ext}}$ according to boundary conditions chosen (force and/or pressure).
7. Linear System Solver: Only for Implicit solvers, used to compute the correction terms, see Appendix B for more details.
8. Update of all vectors

In C++ language which is built on OOP[12], all functions and variables are related to specific classes and rather called methods and attributes. Therefore, the core of the computation is made in the *calculate()* function, which is a method of the 3 inherited classes from class *solvers*. Table 3 specifies on which classes are defined attributes and methods named above (in the code, implicit dynamic is named implicit by default). All of the data needed for the simulation is either stored in *solvers* object or in the arguments given to *calculate()* function.

solvers
• $\_t0/\_tf$ Initial and final times • $\_u/\_v/\_a$ Vectors of displacement, velocity and acceleration • ...
$\star updateMechanicalResponse()$ Update of the internal forces for Implicit solver $\star linearSystem()$ Solving of $A \cdot x = B$ for Implicit solvers $\star$ ...

↙ ↓ ↘

classSolverImplicit	**classSolverImplicitStatic**	**classSolverExplicit**
$\star calculate()$	$\star calculate()$	$\star calculate()$
$\star$ ...	$\star$ ...	$\star$ ...

Table 3: Object-oriented structure of inherited solver classes

[12]Object-Oriented Programming

Concerning the update of $\boldsymbol{F}^{\text{ext}}$, it's done with an external function *NeumannBCManagement()* using a structure called *classNeumannBCs*. All data concerning external forces are stored inside a vector containing as many *classNeumannBCs* objects as defined in the input file. Table 4 shows how all of this objects are built. External forces are applied on elements (at Gauss Points) from t_0 to t_f and can be a constant force, a constant pressure or a ramp pressure.

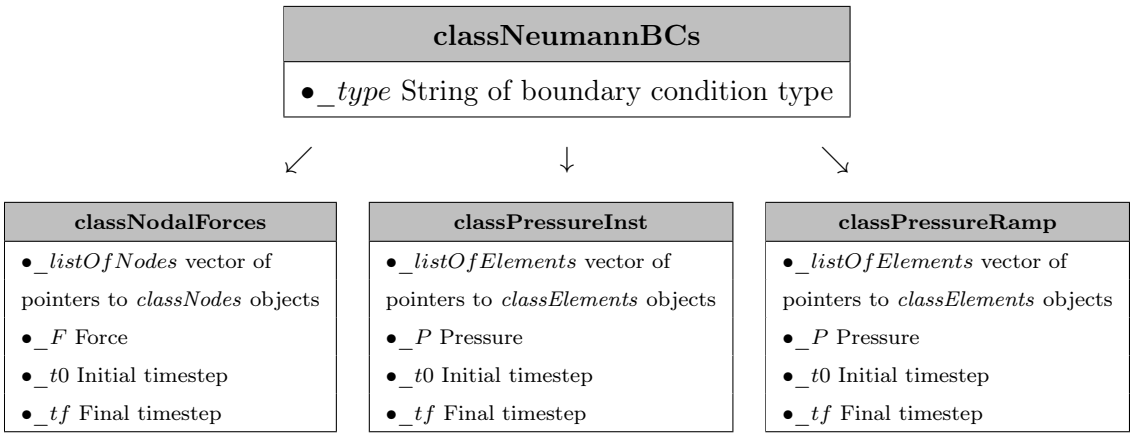


Table 4: Object-oriented structure of inherited classNeuman classes

5.4 Implementation in *OXFEMM*

5.4.1 FHN Model

The implementation of the electrophysiological model –named "extraDof" in *OXFEMM* for "Extra Degree of Freedom"– is made similarly to the mechanical model. All of the data concerning the state of the deformed body are stored for each Gauss point in a vector ***GPs***, where each Gauss point contains the information described in table 6.

classGP
•w Weight of the GP
•J_0 Jacobian at the GP between isoparametric and reference configurations
•$PK1Curr$ First Piola-Kirchhoff tensor
• ...
•$constitutiveModel$ Mechanical model of the element

Table 6: Attributes of object GP (Gauss Point)

The attribute *constitutiveModel* previously only owned a mechanical model such as for example St Venant-Kirchhoff or Neo-Hookean which contain e.g. Young's Modulus, Poisson's ratio, etc. This attribute has been changed such that it's now a vector of *constitutiveModel* objects. If the problem is purely mechanical, this attribute contains only one object which is a mechanical model. But if it is a multiphysical problem, this attribute will contain the address of two objects, two *constitutiveModel*(s), one for mechanics and one for extraDof. If new extraDof models are implemented in the next years, they will be implemented in the same way as the FHN model. Thus, in our case $GPs[i].constitutiveModel[0]$ will contain a mechanical model and $GPs[i].constitutiveModel[1]$ will contain the following FHN model object (Table 7):

classFHN
•Φ_{equi} Equilibrium potential
•Φ'_{equi} 2nd equilibrium potential
•a Activation parameter
•b Activation parameter
•ε Time-scale difference
•I_{stim} Stimulus current
•r Recovery current

Table 7: Attributes of object FHN (FitzHugh-Nagumo model), cf section 5.1

In C++, the header file *FHN.h* contains the structure described above and the source file *FHN.cpp* contains the functions used to update the behaviour of the voltage at the Gauss Point for each timestep: r, R_Φ and $\partial R_\Phi / \partial \Phi$ (only for implicit formulation).

5.4.2 Extra Dof object

Once the FHN model is defined, another class "classExtraDof" is created to store generic extraDof data that are not specific to the model but common to the general Equation (15) such as the diffusion tensor $\boldsymbol{D}$ or the stiffness matrices $\boldsymbol{K}^{\Phi\Phi}$ and $\boldsymbol{K}^{\Phi u}$, even if they are computed differently depending on the extra Dof model. This class also contain all functions needed by each solver to compute each term of the integration scheme, in the same way as the mechanical part, functions *updateMechanicalResponse()* and *NeumannBCManagement()* compute the stiffness matrix $\boldsymbol{K}$ and the force vectors $\boldsymbol{f}^{\text{int}}$ and $\boldsymbol{f}^{\text{ext}}$. Finally, this class allows to compute and get $\boldsymbol{Res}^{\Phi}$ as shown on Table 8.

classExtraDof
• $\boldsymbol{D}$ Conductivity tensor
• $\boldsymbol{\Phi}_n$ Transmembrane potential at current timestep n
• $\boldsymbol{\Phi}_{n-1}$ Transmembrane potential at previous timestep $n-1$
• $\boldsymbol{Res}^{\Phi}$ Residual on electrophysiological equation
• $\boldsymbol{K}^{\Phi\Phi}$ Electrophysiological stiffness matrix
• $\boldsymbol{K}^{\Phi u}$ Electro-mechanical stiffness matrix
• ...
⋆ *DMatrix()* Computation of $\boldsymbol{D}$
⋆ *stiffnessVV()* Computation of $\boldsymbol{K}^{\Phi\Phi}$ (Implicit)
⋆ *stiffnessVU()* Computation of $\boldsymbol{K}^{\Phi u}$ (Implicit)
⋆ *fint()* Computation of $\boldsymbol{F}^{\text{int}}$ (Implicit)
⋆ *fext()* Computation of $\boldsymbol{F}^{\text{ext}}$ (Implicit & Explicit)
⋆ *fintN()* Computation of $\boldsymbol{F}^{\text{int},n}$ (Explicit)

Table 8: Attributes and methods of *classExtraDof* class

5.4.3 Implicit Static and Dynamic solvers

Initialisation In Implicit formulation, the corrective vectors $\Delta\boldsymbol{u}$ and $\Delta\boldsymbol{\Phi}$ (for all nodes) are computed simultaneously with the global stiffness matrix $\boldsymbol{K}^{\text{total}}$ defined in Equation (34). The linear system to solve at each Newton-Raphson iteration is thus the following:

$$\begin{pmatrix} \boldsymbol{K}^{uu} & \boldsymbol{K}^{u\Phi} \\ \boldsymbol{K}^{\Phi u} & \boldsymbol{K}^{\Phi\Phi} \end{pmatrix} \cdot \begin{pmatrix} \Delta\boldsymbol{u} \\ \Delta\boldsymbol{\Phi} \end{pmatrix} = \begin{pmatrix} \boldsymbol{Res}^{u} \\ \boldsymbol{Res}^{\Phi} \end{pmatrix} \Longleftrightarrow \boldsymbol{K}^{\text{total}} \cdot \Delta\boldsymbol{Dof}^{\text{total}} = \boldsymbol{Res}^{\text{total}} \tag{44}$$

with $\boldsymbol{Res}^{u}$ and $\boldsymbol{Res}^{\Phi}$ the vectors of all mechanical and extraDof residuals. If Implicit Dynamic solver is used, $\boldsymbol{Res}^{u} = \boldsymbol{r}$ and $\boldsymbol{K}^{uu} = \boldsymbol{K}^{eq}$, otherwise $\boldsymbol{Res}^{u} = \boldsymbol{Res}$ and $\boldsymbol{K}^{uu} = \boldsymbol{K}$ (cf section 4.2). Finally, the following new vectors, matrices or objects are needed for the implementation and declared at the beginning:

- vvK: vector $\mathbf{\Phi}$ on all nodes
- deltaExtraActiveDof: $\Delta\mathbf{\Phi}$ on only active[17] nodes
- resExtraActiveDof: $\boldsymbol{Res}^{\Phi}$ on only active nodes (not Dirichlet nodes)
- resExtraDof: $\boldsymbol{Res}^{\Phi}$ extended to all nodes
- deltaDof_total: $\Delta\boldsymbol{Dof}^{\text{total}}$ on only active DOFs (mechanical and electrophysiological)
- res_total: $\boldsymbol{Res}^{\text{total}}$ on only active DOFs (mechanical and electrophysiological)
- Kuv, Kvu, Kvv and KK_total: $\boldsymbol{K}^{u\Phi}$, $\boldsymbol{K}^{\Phi u}$, $\boldsymbol{K}^{\Phi\Phi}$ and $\boldsymbol{K}^{\text{total}}$ on only active DOFs
- extraDof: "classExtraDof" object

In addition to that, doubles "normResExtra" and "tolResExtra" are created and initialised in the same way as in the mechanical part. The convergence criteria is changed such that the Newton-Raphson loop is stopped only if there is convergence on both mechanical and extra Dof residuals.

Time Loop There is no static or dynamic prediction at the beginning of the time loop, $\mathbf{\Phi}_{n+1}$ is initialised with the previous vector $\mathbf{\Phi}_n$. Dirichlet boundary conditions are applied on the corresponding nodes.

Newton-Raphson Loop Attributes $\boldsymbol{Res}^{\Phi}$, $\boldsymbol{K}^{\Phi u}$ and $\boldsymbol{K}^{\Phi\Phi}$ of extraDof object are updated with the current version of $\mathbf{\Phi}_{n+1}$. If the norm of $\boldsymbol{Res}^{\Phi}$ vector is higher than a given tolerance, the processor begins Newton-Raphson's iterations until having the $\|\boldsymbol{Res}\mathbf{\Phi}\|_2 \simeq 0$.

To solve global Equation (44), all concerned matrices and vectors are concatenated into "total" vectors as defined in the initialisation paragraph. To solve this linear system, *OXFEMM* developers choose to develop a *linearSystem()* function using a library named PETSC that give the solution with tolerance factor. After using this solver, the solution vector $\Delta\boldsymbol{Dof}^{\text{total}}$ is separated into its mechanical and extraDof parts. The correction term $\Delta\mathbf{\Phi}$ is added to the voltage vector $\mathbf{\Phi_{n+1}}$, and the extraDof residual $\boldsymbol{Res}^{\Phi}$ is computed again until reach convergence. Figure 10 shows the algorithm implemented for Implicit Dynamic Solver.

5.4.4 Explicit solver

Initialisation In Explicit formulation, the following new vectors, matrices or objects are needed for the implementation and declared at the beginning:

- vvK: vector $\mathbf{\Phi}$ on all nodes
- rhsExtra: $\boldsymbol{Res}^{\Phi}$ on all nodes
- massMlumpedExtra: $\boldsymbol{M}^{\text{extra}}$ matrix lumped into $\boldsymbol{M}^{\text{extra}}_{\text{lump}}$ vector
- extraDof: "classExtraDof" object

Time Loop Dirichlet boundary conditions are applied on the corresponding nodes. Attribute $\boldsymbol{Res}^{\Phi}$ of extraDof object is updated with the previous version of $\mathbf{\Phi}_n$ and allows to compute the prediction of $\mathbf{\Phi}_{n+1}$ vector for the next time step (cf Equation (43)).

[17] Active means that there is no any Dirichlet boundary condition on this DOF or node

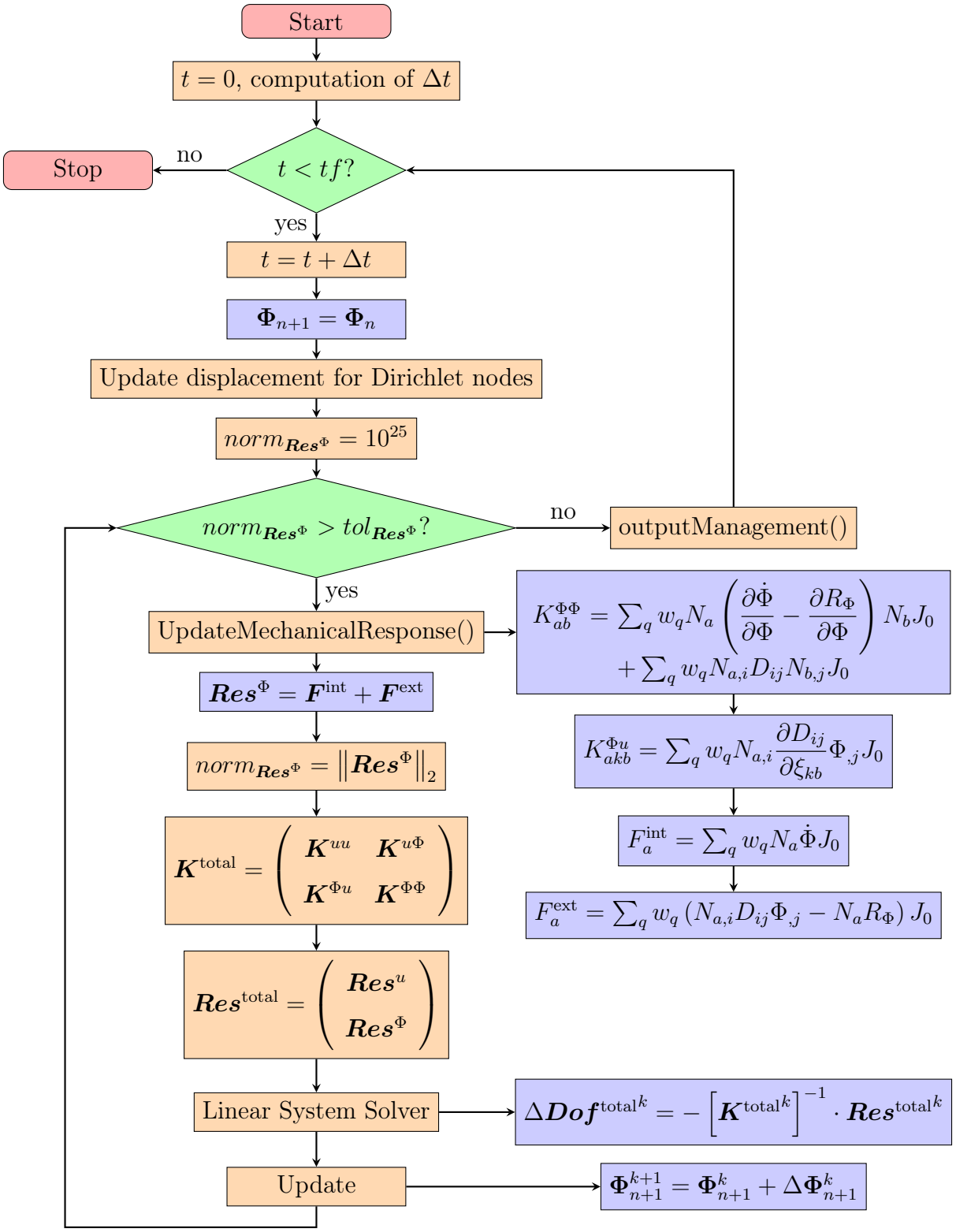


Figure 10: Implicit Dynamic solver algorithm for electrophysiological formulation