Latin American Journal of Solids and Structures
www.lajss.org

An Object-Oriented class design for the Generalized Finite Element Method programming

Abstract

The Generalized Finite Element Method (GFEM) is a numerical method based on the Finite Element Method (FEM), presenting as its main feature the possibility of improving the solution by means of local enrichment functions. In spite of its advantages, the method demands a complex data structure, which can be especially benefited by the Object-Oriented Programming (OOP). Even though the OOP for the traditional FEM has been extensively described in the technical literature, specific design issues related to the GFEM are yet little discussed and not clearly defined. In the present article it is described an Object-Oriented (OO) class design for the GFEM, aiming to achieve a computational code that presents a flexible class structure, circumventing the difficulties associated to the method characteristics. The proposed design is evaluated by means of some numerical examples, computed using a code implemented in Python programming language.

Keywords
Object-Oriented programming; Generalized Finite Element Method; Python programming language

Dorival Piedade Neto[*]
Manoel Dênis Costa Ferreira
Sergio Persival Baroncini Proença

Department of Structural Engineering, School of Engineering at São Carlos,
University of São Paulo, São Carlos, Brazil

Received 19 Jan 2013
In revised form 26 Feb 2013

[*]Author email: dpiedade@sc.usp.br

1 INTRODUCTION

The Generalized Finite Element Method (GFEM) is a numerical approach that explores the partition of unity (PU) framework to generate enriched approximated solutions for differential equations. Similarly to the Finite Element Method (FEM), the problem domain is described by means of a mesh of elements, defined by discrete points named nodes, for which the numerical solution is searched. Each node of the mesh can be considered the vertex of a cloud, defined by the set of elements sharing it. The GFEM shape function is then constructed by performing the product between the partitions of unity shape functions and an adopted enrichment function, resulting in an extra degree of freedom for the vertex node. The enrichment function can be a polynomial or any generic function, which turns the method powerful to solve problems for which the solution characteristic is known, since the enrichment function can be chosen to fit with the local characteristic of the differential equation solution.

The support of the GFEM shape function defines a region named cloud, in which the enrichment function is applied. Due to such characteristic, it is possible to apply 'enrichments' only in specific vertex nodes of the domain. This local enrichment feature, usually referred as selective enrichment refinement, represents a more flexible numerical strategy, when compared to the h and p-refinement of the conventional FEM, which allows one to achieve better results without modifying the mesh topology or inserting additional nodes in the mesh, see Duarte and Oden (1996a) and Duarte and Oden(1996b).

Despite of these advantages, the GFEM computational implementation can be considered notably more complex than the FEM implementation. One of the main reasons for that is the fact that the number of degrees of freedom for each node varies according to number of enrichment functions applied to it. In addition, in order to fully achieve the GFEM benefits, it is important to have a framework capable to deal virtually with any kind of enrichment function, or at least, that allows one to easily add new enrichment functions to the code. Yet, another GFEM characteristics demand a computational data structure substantially more difficult to conceive, and turn the method specially beneficiated by the Object-Oriented Programming (OOP) paradigm.

The OOP application for finite element analysis codes started to be presented in the technical literature in the beginning of the 1990s, see for instance, Forde et al. (1990) and Alves Filho and Devloo (1991). In the first paper, the basic concepts of the OOP are introduced, followed by a description of an OO design for the FEM. The described OO code is also compared to an equivalent procedural program, identifying the advantages of the OO approach.

Such works were followed by Zimmermann et al. (1992) and Dubois-Pélerin et al. (1992), in which the paradigm is described in more details using Smalltalk programming language. Next it was published a paper describing an efficient OO implementation using C++, see Dubois-Pélerin et al. (1993). The use of advanced features of C++ to implement finite element classes is presented later in Bittencourt (2000).

Many other papers on the use of the OOP for numerical analysis were published, like, for instance, Mackie (2000), and the subject has been intensively discussed along the last decades. Skipping from a detailed review of the OOP for FEM, which is not the focus of the present paper, w just refer to Mackerle(2000), which lists hundreds of references on the subject, also including a list of OOP applications to the Boundary Element Method.

The application and detailed description of the OOP for other variants of non-conventional numerical methods is clearly limited in the available literature. For instance, among the works reviewed regarding exclusively to the GFEM, its implementation by OOP was found by the authors only in one dissertation, Pereira (2004), written in Portuguese, and in which one may find a description of one possible structure of classes composing this kind of computational framework. Regarding technical periodicals, the only reference addressing to a closer method to the GFEM, the eXtended Finite Element Method (XFEM), is presented by Bordas et al(2007). Even thought important contributions are given in such works, none of them presents a detailed description of the class structure to efficiently support the GFEM analysis, circumventing the difficulties addressed before.

The present paper is devoted to describe an object-oriented (OO) class structure designed to deal with the GFEM characteristics, resulting in a computational code capable to take advantage of the method's flexibility and power. Departing from a conventional FEM general class structure, new classes and features are inserted in such framework. The code design description is complemented by a discussion about the adopted programming language. We advocate the use of Python programming language to implement such framework. The previous statement is technically justified by the fact that its dynamic nature fits perfectly with the problem characteristics, as it is shown in the present text.

On what follows, in section 2 the main features of the Generalized Finite Element Method are addressed, underlining its flexibility and the main difficulties faced to implement it computationally, aiming to perform two-dimensional linear structural analysis. In section 3 the OOP paradigm is briefly addressed, and the class structure designed to implement the GFEM framework is described. Section 4 is devoted to present numerical example to evaluate the flexibility and accuracy of the resulting code. Also the resulting code time performance is evaluated. Next, in section 5, it is presented a brief discussion on how the presented OO design can be extended to support nonlinear solid mechanics analysis. Also some illustrative examples of nonlinear analysis, performed using the resulting code, are presented. It turned out that the presented structure is indeed flexible and suitable enough to be used as the basis for building a complete nonlinear analysis framework. Finally, a balance among advantages/disadvantages is presented in section 6, concluding that the presented class structure and the advocated programming language (Python) are convenient for the development of a general purpose GFEM analysis code.

2 THE GENERALIZED FINITE ELEMENT METHOD

The Generalized Finite Element Method, Duarte and Oden(1996a), is based on the concept of enriched partition of unity (PU), see Melenk and Babuška (1996). Formally, a PU is mathematically defined as a set of compact support, smooth and continuous functions φ_i such that:

$$\sum_{i\in I}\varphi_i\left(\xi\right)=1, \tag{1}$$

$$\varphi_i:\xi\rightarrow\left[0,1\right]. \tag{2}$$

Basically, a set of functions defined in a certain domain constitutes a PU if its sum is equal to one for all points in the domain, as indicated in equation (1). In addition to it, all functions must present values in the closed interval from 0 to 1 of its compact support,according to equation (2).

Partitions of unity are used in the Finite Element Method (FEM) to interpolate function fields from values at discrete points of the continuum attached to nodes, extending the local value approximations to the whole domain. The linear Lagrange polynomials set, used in the classical displacement based finite elements, is an example of PU, and in general, these polynomials are the ones used in the GFEM implementation.

Once a mesh of Lagrangian elements is defined, the generalized interpolations can be constructed by multiplying the PU to polynomials or any other special purpose functions. These shape functions are then employed according to the Galerkin method to compute approximate solutions to boundary value problem, as it happens in the standard FEM. Actually, as it is shown next, the procedure adopted to construct the shape functions allows that the same computational framework can be used to implement both FEM and GFEM codes.On what follows one focus on the main features of the GFEM framework keeping in mind that the code herein described is restricted to two dimensional linear solid mechanicsanalysis.

The weak form of the boundary value problem can be obtained using the Principle of Virtual Work, being represented as:

$$\int_{\Omega} \boldsymbol{\sigma} : \boldsymbol{\delta\varepsilon}\, d\Omega = \int_{\Gamma_\sigma} \boldsymbol{p} \cdot \boldsymbol{\delta u}\, d\Gamma + \int_{\Omega} \boldsymbol{b} \cdot \boldsymbol{\delta u}\, d\Omega, \forall \boldsymbol{\delta u}. \tag{3}$$

The left hand side of equation (3) is the internal virtual work given by the product of the stress field $\boldsymbol{\sigma}$ and the virtual strain field $\boldsymbol{\delta\varepsilon}$ in the solid domain, while the right hand side represents the external virtual work, in which $\boldsymbol{p}$ is the vector of forces applied at the solid boundary (*i.e.*, Neumman boundary condition), $\boldsymbol{b}$ is the vector of body forces and $\boldsymbol{\delta u}$ is the virtual displacement field.

Once the Galerkin approximations provided by the GFEM are adopted to approximate the boundary value problem, the global system of equation is assembled by the contributions of the element or local stiffness matrices, each one computed by the following expression:

$$\boldsymbol{K}_l = \int_{\Omega_e} \boldsymbol{B}^T \boldsymbol{D} \boldsymbol{B} d\Omega. \tag{4}$$

In equation (4) $\boldsymbol{B}$ is a matrix containing the partial derivatives of the approximation field, and relates the strain tensor $\boldsymbol{\varepsilon}$ to the displacement field $\boldsymbol{u}$, while $\boldsymbol{D}$ is the constitutive stiffness matrix, relating the stress tensor $\boldsymbol{\sigma}$ to the $\boldsymbol{\varepsilon}$ tensor. In a similar fashion, the global force vector is also derived from the external virtual work term. One can observe that the assembling procedure is essentially analogous to the one described in the classical FEM literature, as for instance, Hughes (2000) or Bathe(1996).

As already mentioned before, the GFEM explores the FEM mesh to build its shape functions. The 'enriched' shape functions are obtained by multiplying the PU shape functions by polynomial or special purpose functions. These functions are generally referred as enrichment functions, since they are used to achieve a better field interpolation. Each of these shape functions is associated to a given vertex node a, to which a region ω named cloud is attached. In the general sense the cloud is the region in which the shape function presents nonzero values, *i.e.*, the region of its support, being defined by the set of elements containing the vertex as a common node.

A GFEM cloud for the two dimensional case is depicted in Figure 1. The cloud radius is defined as the radius h of the circle surrounding the cloud's elements. The cloud radius is introduced as a dimensional factor in the enrichment functions for avoiding numerical conditioning problems in the system of equation.

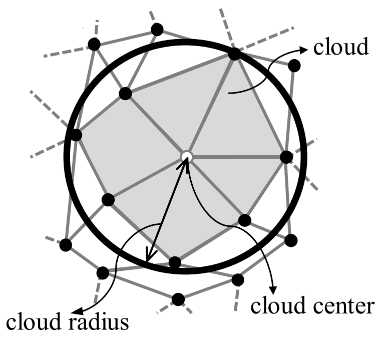


Figure 1 A node's cloud and its radius, in a two dimensional domain.

The PU function of a given cloud a is represented by the union of the nodal shape functions of each element attached to the common vertex node. In order to achieve the enriched shape function, it is usual to employ a (n_{enr}x*1*) column matrix $\boldsymbol{L}$, containing n_{enr} enrichment functions L^{enr}. It is also common that the first component of the $\boldsymbol{L}$ matrix is equal to *1* (one). This is useful in order to keep apart the regular interpolation of the FEM and the additional enriched interpolation terms. Thus, the regular GFEM approximation preserves the FEM fields while including additional enriched interpolation terms, resulting in:

$$u^{aprox.} = \left(\sum_{i=1}^{n_c} \varphi_i u_i\right)\boldsymbol{L} = \sum_{i=1}^{n_c} \varphi_i u_i + \sum_{i=1}^{n_c}\sum_{j=2}^{n_{enr}} \varphi_i L_j^{enr} u_{ij} = \underbrace{\sum_{i=1}^{n_c} \varphi_i u_i}_{regular\,interpolation} + \underbrace{\sum_{i=1}^{n_c}\sum_{j=2}^{n_{enr}} \varphi_{ij}^{enr} u_{ij}}_{enriched\,interpolation} . \tag{5}$$

In the previous relation, n_c is the number of clouds, u_{ij} are the additional nodal parameters in correspondence to each one of the enrichment function, φ_i represents the regular PU and n_{enr} is the number of enrichment functions.

In general there are no restrictions on the choice of the enrichment function, so one may use any type of function. In the present work we have employed polynomial functions aiming at results similar to the ones achieved by means of higher order partitions of unity. These functions are defined by means of the Cartesian coordinate system variables, and they are evaluated according to the distance to the cloud center. An example of a set of polynomials functions to be used in a complete 'second degree enrichment' is

$$\boldsymbol{L} = \left\{1, \frac{(x-x_\alpha)}{h}, \frac{(y-y_\alpha)}{h}, \frac{(x-x_\alpha)^2}{h^2}, \frac{(x-x_\alpha)(y-y_\alpha)}{h^2}, \frac{(y-y_\alpha)^2}{h^2}\right\}. \tag{6}$$

In this paper these functions are referred as 'bubble like functions', since they are equal to zero in the cloud's center (x_a, y_a). This characteristic is useful to preserve the meaning of the original nodal parameters once the enrichment is adopted. Moreover, it becomes easier to impose Dirichlet boundary conditions in nodes enriched only with functions like these ones.

If one intends to use a first degree polynomials set, only the three first terms of (6) must be used; for higher degrees, the set is generated according to the degrees of the Pascal's triangle polynomial expansion.

Aiming to use the same computational framework both for the FEM and the GFEM, one may also compute the local GFEM stiffness matrices by means of an element by element systematic assembling procedure, according to equation (4). However, for the GFEM, the $\boldsymbol{B}$ matrix order is dependent on the enrichment set $\boldsymbol{L}$ adopted. Then, the resulting local matrix contains more terms than the original one for no enrichment status. Actually, additional lines and columns appear in correspondence to the new nodal parameters introduced by the enrichment fields. Figure 2 illustrates some schemes of the local stiffness matrix for a displacement based linear triangular element in correspondence to different enrichment possibilities. In the FEM, each node has two degrees of freedom. If both degrees of freedom of one of its nodes is enriched with a $\boldsymbol{L}=\{1, Le\}$, the original (6x6) matrix expands to a (8x8) shape matrix. If the same enrichment is applied to all the nodes, it results an (12x12) matrix.

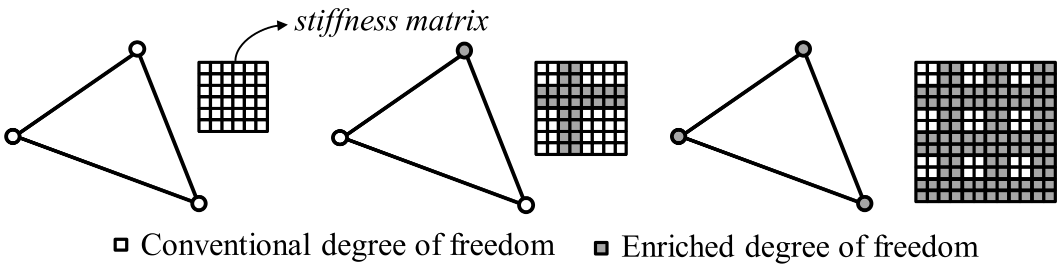


Figure 2 Local stiffness matrix form according to the enrichment used in its nodes.

Even though the basic idea seems quite simple, its implementations in not straightforward, requiring a substantially complex algorithm design.The efforts demanded in order to implement it are justified by the fact that, in general, the GFEM demands much coarser meshes than the ones required by the FEM. On the other hand, the computing of the enriched element stiffness matrix is usually more time expensive, demanding a well designed computational code, in order to obtain a flexible and yet efficient computational code. The main actor in this framework is the element class method, which must be able to build generic $\boldsymbol{B}$ matrices. The proposed design allows one to implement methods to compute such matrix in a generic and yet efficient fashion.

3 FEM AND GFEM OBJECT-ORIENTED DESIGN

The Object-Oriented Programming (OOP) is a programming paradigm in which one creates abstract models by defining data structures named objects. Objects are data instances defined by means of classes, which enclose the description of the internal variables describing a category of objects, plus the functions and procedures relating to it, referred to as methods. Such objects may be used to represent real world objects or even theoretical model entities. The objects' methods are used to change its own variables or to send messages to other objects, requesting data or even asking other objects to perform tasks. Following such strategy, the computational program is then executed by means of a set of different classes' objects, interchanging messages in order to perform the tasks requested by the software user.

Since the concept of class provides a manner of grouping variables, constants and methods altogether in the same data structure, the OOP provides a logical framework in which the data manipulation is restrict to the module in which the data is defined. This is essentially the attribute of encapsulation, which provides modularity to the code. One consequence of encapsulation is that a detailed knowledge about the internal implementation of the module is no longer necessary to the external actors, and only the definition of its public method interface suffices for using it. If the module is designed properly, it can be used in different situations in the same manner by means of the same interface, providing flexibility and re-usability of the code.

Such as for other programming paradigms, for the OOP there is not a unique way to create the computational code. For each adopted design, different characteristics are obtained. Different performance results (both for processing time and memory storage), code flexibility, readability and many other characteristics can be observed in the resulting code. For instance, the OOP design is known for its modularity and expansibility, specially desired characteristics to perform collaborative programming. Many other advantages of the OOP can be stated, but we refer to Cross et al(1999) which advocates the use of such paradigm for finite element analysis codes.

Nevertheless each developed code present specific characteristics, some data structures are basically the same for any code of a given type of software, and so, it is possible to define a general class design for it. For instance, Figure 3 depicts a generic class design for a finite element linear analysis code.

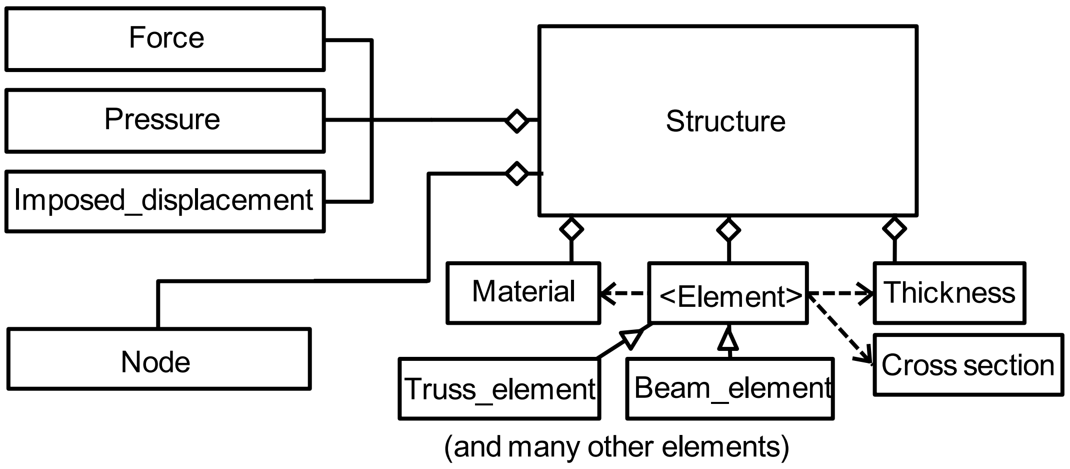


Figure 3 General FEM object-oriented design.

In such scheme, the diamond symbol in the connecting lines indicates a composition, *i.e.*, one class instance contains instances of the other class. For instance, the structure may contain instances of nodes, materials and so on.

The triangle symbol indicates a relation of inheritance, *i.e.*, that one class derives from another. The inheritance is observed when a specific type of object is a subtype of a generic object. This is the case of the 'beam element', for instance, which is a type of element, and so, inherits some common methods and variables from the generic element class. It is important to note that in such case, the generic element does not exist in practice, *i.e.*, one cannot instantiate an object of such class. This is called an abstract class, and it is represented in Figure 3 putting its name between angle brackets (<Element>).

Finally, it is important to explain the meaning of the dashed line arrows that connect the element class to the 'material', 'thickness' and 'cross section' classes. As it can be observed, instances of such classes are instantiated inside the 'structure' object (observe the diamond symbol in Figure 3). It is clear that such objects are used to define different types of materials, cross sections and thicknesses to describe a set of elements, and that in general, several elements share the same material and geometric characteristics. It would be a waste of memory instantiate one instance of them for each of the elements of a structure. So, one just needs to define the materials, thickness and cross sections objects once, which are stored in the structure instance, and then, in order to associate an element to such object, it is just necessary to point to such material or thickness instance. This is the meaning of the dashed line arrows: it indicates that the element's material type and cross section data, for instance, is defined by pointing to the previously defined material and cross section instances, which are stored in the structure object.

Regarding to the generic FEM OO design of Figure 3, within such scheme, the node class is one of the main actors, representing the discrete points of the continuum at which the desired result is computed. Therefore, each node is represented by its Cartesian coordinates and must store the results achieved by means of the numerical method.

The continuum regions between the nodes constitute finite dimension elements, which define another fundamental class in the FEM/OOP approach. Beyond the set of nodes defining it, the elements instances must also hold information about the material and additional geometric characteristics, such as its thickness (plane and shell elements) or cross section data (beam and truss elements). As already mentioned, such characteristics are defined by means of additional classes as 'thickness'/'cross section', which also constitute the data structure presented in Figure 3.

Also boundary conditions data are necessary to define the problem, and so classes to describe them are demanded. For solid mechanics problems using displacement based formulation, classes like 'force', 'pressure' and 'imposed displacement', for instance, are related to the boundary conditions.

This basic class framework design is completed by the 'structure' class. The structure is the main instance in such data structure, and constitutes the main interface to the rest of the code. In fact, it is used to define the finite element model characteristics and to manage the other classes' instances. Nevertheless, depending on the desired usage for the code, a more general name for such class is 'structural set' or 'structural problem', which makes sense if one thinks about problems treating a set of solids, as, for instance, the solid mechanics contact problems. Departing from the general OO design (Figure 3), the modifications demanded in order to comport the GFEM are presented in the following section.

3.1The OOP for the GFEM

The major changes in the present FEM structure in order to support the GFEM are related to the node class. As already discussed previously, in the GFEM, the number of degrees of freedom (DOF) associated to a given node varies according to the number of enrichment functions applied to it. In addition, for all computations, each specific enrichment data must be directly associated to the new DOF.

In fact, it turns out that the enrichment function, which does not even exists for the FEM, is an important actor within this framework. Furthermore, given its generic nature, such 'abstract' entity can be better described by means of defining an abstract <enrichment> class, the basis for a polymorphic set of different enrichment classes.

By taking all these aspects into account, in the proposed OO design a set of classes replaces the FEM node class, as indicated in Figure 4.

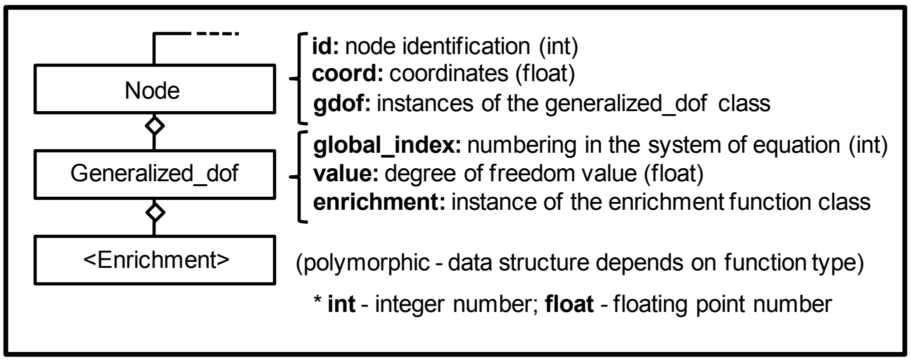

Node
id: node identification (int)
coord: coordinates (float)
gdof: instances of the generalized_dof class
Generalized_dof
global_index: numbering in the system of equation (int)
value: degree of freedom value (float)
enrichment: instance of the enrichment function class
<Enrichment>
(polymorphic - data structure depends on function type)
* **int** - integer number; **float** - floating point number

Figure 4 Node, generalized degree of freedom and enrichment class.

In order to associate a new nodal degree of freedom with enrichment function instances, we propose the definition of a 'generalized degree of freedom' class. Regarding its functionality, this class basically holds the scalar value of the degree of freedom, computed by means of the numerical method, its numbering in the system of equations (global_index), and an instance of the 'enrichment function' class, if any enrichment is applied to such DOF. If no enrichment is applied, such instance must be null, resulting in a regular FEM degree of freedom. It is important to mention that for the GFEM, due to the variable number of DOF for each node, the numbering association to the system of equations is fundamental to perform the element stiffness matrices and element load vectors contributions in the global system of equations.

Since the new data structure for the node class must be able to hold a variable number of generalized degrees of freedom, it demands intrinsically the usage of a dynamic data structure, like, for instance, the linked lists. Even though linked lists can be constructed in any programming language by using pointers and user defined data types, we have explored the use of native dynamic list provided by Python programming language, avoiding extra work in such data structure implementation.

Python is a multiplatform programming language, being considered by some authors, such as Tucker et al(2008), a multi-paradigm language, which supports imperative and object oriented programming paradigms. It is a script language, which means that the resulting code is interpreted in the run-time and not compiled, as required by Fortran or C languages, for instance. This fact leads to some advantages and some possible drawbacks.

According to Langtangen(2008), the higher abstraction level inserted in scripting can turn programming more convenient. Actually, scripting languages allow for connecting different applications, as scripts are efficient in receiving inputs and formatting them to outputs to another application. Such use of Python is applied by Layman et al (2008). In fact Python is nowadays being used by many Finite Element Packages such as Abaqus, as reported by Kuutti and Kolari (2012).

On the other hand, one of the main concerns with script languages, especially for numerical computation applications, is the performance issue. However, thanks to the existence of a set of numerical support libraries, like NumPy, TheSciPy Community (2010), SciPy, The SciPy Community(2011), and Matplotlib, Dale et al(2011), Python can be used to develop efficient numerical applications. In fact, the performance of the developed code is shown in section 5, and proved to be sufficient for the purposed application.

Even though a GFEM code could be developed in any other programming language, the main Python characteristic that defined it as the programming language to develop the proposed OO framework is its dynamic native data structure, which fits exactly with the nature of the generalized degrees of freedom sets of the GFEM. The previous statement becomes clear in the next paragraphs, in which other native Python data structures are applied to compose the proposed classes.

Turning back to the OO design, the 'node' class demands a data structure that is capable to hold any type of degree of freedom, such as displacement, temperature or magnetic field, for instance, and for each of them hold any number of generalized DOF instances.

In order to do so, the proposed design uses a Python dictionary named 'gdof'. The dictionary is another native Python data structure in which 'keys' (in practice, any immutable Python object) are associated to any other type of Python objects, including mutable native Python variables or even user defined data types. The 'gdof' dictionary keys represent the type of DOF, while the related value is a Python list in which one can insert any number of 'generalized degree of freedom' instances, depending on the number and type of enrichment defined by the user. Such scheme is indicated in Figure 5.

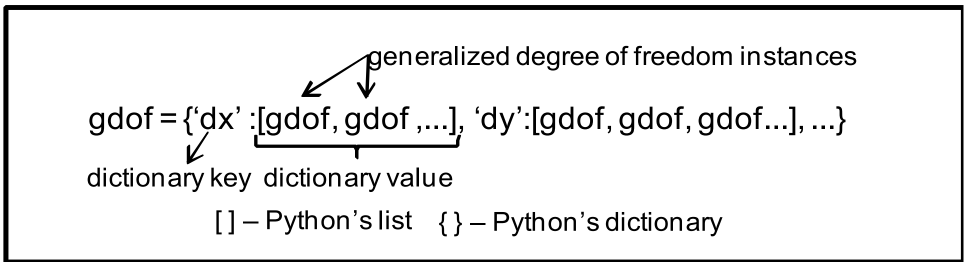

generalized degree of freedom instances
gdof = {'dx' :[gdof, gdof ,...], 'dy':[gdof, gdof, gdof...], ...}
dictionary key
dictionary value
[ ] – Python's list
{ } – Python's dictionary

Figure 5 The nodal generalized degree of freedom data structure.

In Figure 5, 'dx' and 'dy', for instance, stand for 'displacement in x direction' and 'displacement in y direction', respectively. As already mentioned, this allows one to include any other type of DOF in such data structure. Specifically, we find out in our implementation that such tool is efficient to hold Lagrange multipliers variables used for imposing displacements efficiently in the GFEM.

The proposed data structure allows that the generalized element's stiffness matrix and load vector computations still being performed by means of the same strategy of the traditional FEM, i.e., by applying equation (4). The enriched terms contributions is taken into account in the routine that computes the $\boldsymbol{B}$ matrix, in which it is verified whether the generalized degree of freedom is enriched or not. In none of the DOF of a given element is enriched, the FEM $\boldsymbol{B}$ matrix is obtained.

Finally, an important aspect towards a generic framework is the fact that the enrichment function is represented by a set of polymorphic classes. Such an abstract representation allows that virtually any type of function can be included in the code. This advantage is especially convenient for scripting languages, like Python, since one can include new enrichment function without needing to compile the code again.

Another important aspect focused in the proposed class design is related to the shape functions used as partitions of unity. As discussed previously, if one follows strictly the definition of partition of unity, only the linear (triangular) or bi-linear (quadrilaterals) Lagrangian shape functions can be used as PU for the GFEM. In such context, polynomial enrichments functions are fully justified since such basic PU's generally provide poor quality results. In fact, these PU often demand a hierarchical strategy (h-refinement) for the traditional FEM, resulting in refined meshes in order to achieve an accurate result. A comparison on the efficiency of GFEM-refinement and conventional h-refinement strategies is presented in section 4, justifying the convenience of the polynomial enrichment for the GFEM.

On the other hand, even though the bubble like enrichment increases significantly the quality of the results, as it is verified in section 4, the use of linear isoparametric elements in order to accurately describe the geometry of curved boundary solids is not possible. So, within the present context, the possibility to have linear PUs associated to curved geometry elements is very relevant. In order to do so, the proposed OO design solution is based on the definition of a polymorphic class named 'Partition of Unity', basically defining generic PUs of any desired domain, which can be one-dimensional, two-dimensional (triangular or quadrilateral) or even three-dimensional.

Even though it might seem abstract to have a generic shape function class, such idea allows one to employ different approximation PUs to describe the elements geometry and the physical behavior (field interpolation) by defining different PU instances inside the element data structure, as indicated in the Figure 6. As it can be noticed, by adopting this design, both isoparametric FEM elements and curved geometry linear GFEM PU can be supported using the same code framework.

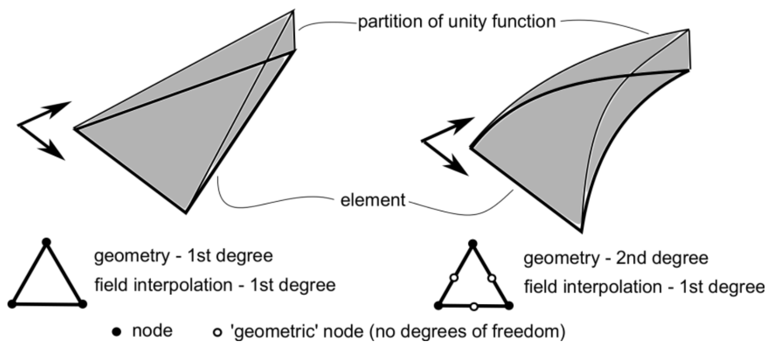

partition of unity function
element
geometry - 1st degree
field interpolation - 1st degree
geometry - 2nd degree
field interpolation - 1st degree
● node
○ 'geometric' node (no degrees of freedom)

Figure 6 Different PU approximation to describe the geometry and the field interpolation.

When considered altogether, the hereby proposed set of changes in the classical FEM OO class design allows that basically the same framework can be efficiently extended to the GFEM purposes, then preserving the generality of the original methods. In fact, the other classes presented in Figure 3 remain practically the same, requiring at most little or even no changes in order to support both FEM and GFEM models. Figure 7 presents the complete proposed OO class design for the GFEM.

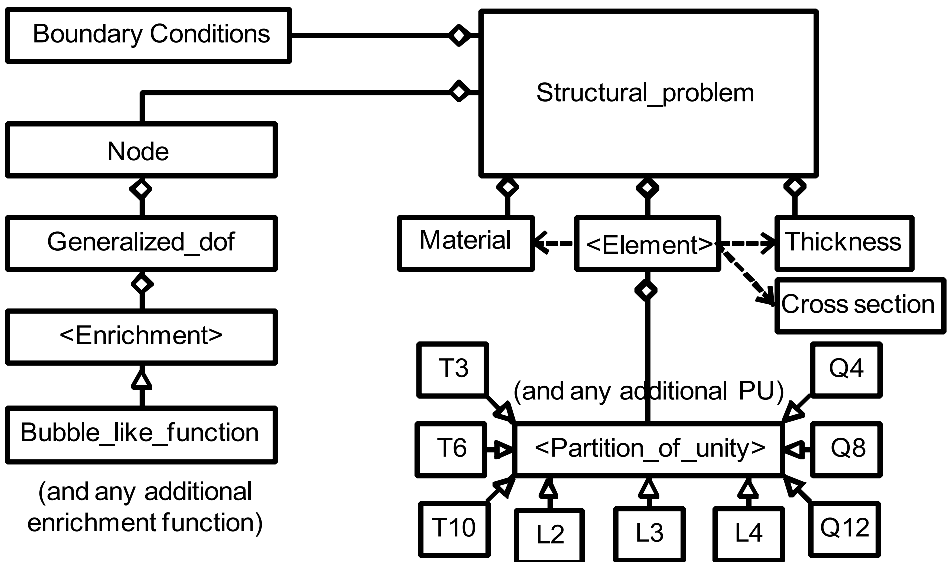


Figure 7 The proposed object-oriented design for the GFEM.

In Figure 7, T3, T6 and T10 are triangular domain two-dimensional PUs, while Q4, Q8 and Q12 are quadrilateral PUs. The L2, L3 and L4 classes are one-dimensional PUs. The numbering in their names makes reference to the number of nodes needed to define such PU.

As it can be noticed we also propose the union of the force, pressure and imposed displacement classes in a single class named ‘boundary condition’, searching to achieve a more generic representation of such data. This unification is a straightforward modification and present no major details to be described, but improves significantly the code generality, especially if one aims to employ the same framework not only to solid mechanics analysis code, but to solve other types of partial differential equation problems.

4 NUMERICAL EXAMPLES

In order to test the computational code accuracy and to evaluate its flexibility and capability of treating both the traditional finite element and the generalized finite element analysis, next follows a simple cantilever beam and a solid with a hole examples, both modeled under plane stress hypothesis. Then, a last example to evaluate Python’s processing time performance is presented.

4.1 Cantilever beam

A 24.0x6.0x0.1 (dimensionless) cantilever beam subjected to a distributed load q=1.0 is stated, as illustrated in Figure 8.

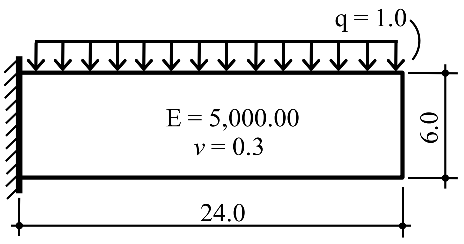


Figure 8 Cantilever beam scheme.

In order to verify the accuracy in the conventional isoparametric finite element formulation implemented in the code, the structure is modeled by means of quadrilateral regular meshes containing 16 elements (bi-linear, bi-quadratic and bi-cubic). The horizontal stress component achieved results are indicated in Figure 9.

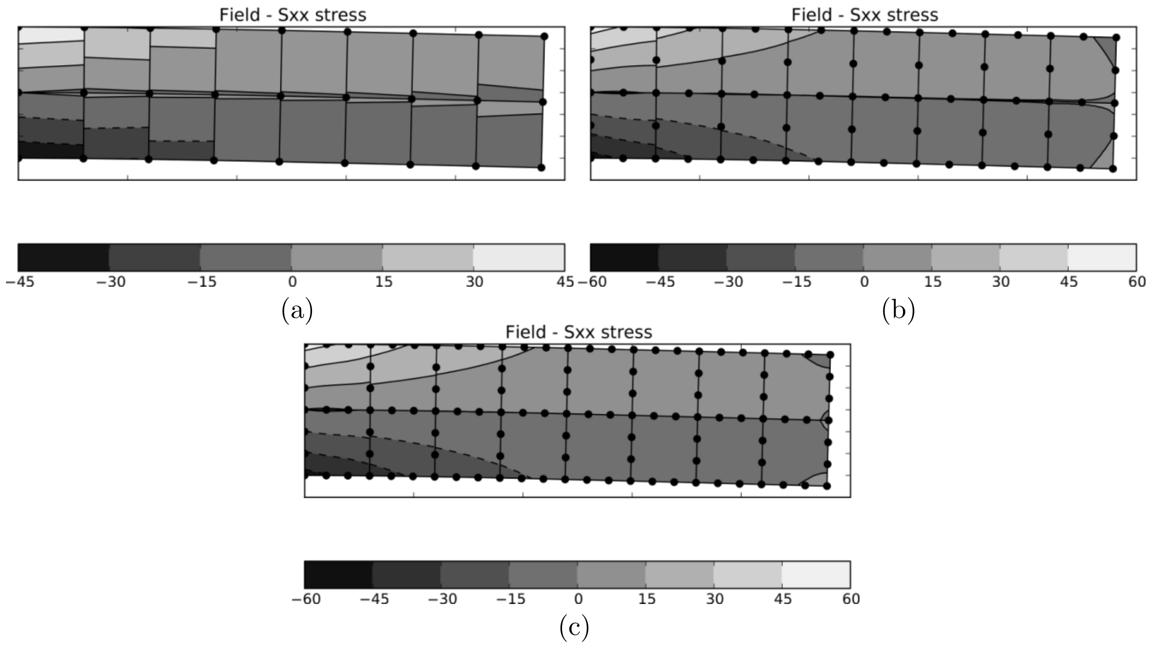


Figure 9 Horizontal stress component field map achieved employing bi-linear (a), bi-quadratic (b) and bi-cubic (c) isoparametric elements.

In order to evaluate the GFEM implementation, the same mesh of bi-linear elements is used. Appling a second degree enrichment in all its nodes (except the ones in which the Dirichlet boundary condition is applied), one observe a stress field map close to the ones achieved for the higher order isoparametric finite elements (see Figure 10).

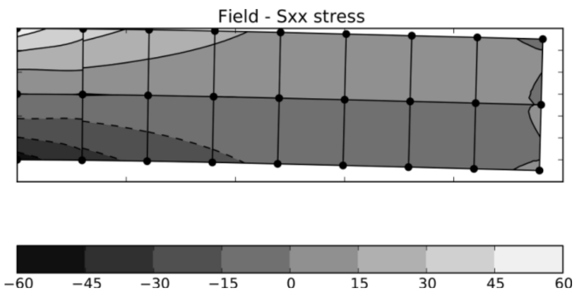


Figure 10 Horizontal stress component field map for a second order bubble like enrichment applied in a bi-linear element mesh.

Since the developed code supports both FEM and GFEM models, the stated example can be used to perform a comparison among the polynomial (p) and hierarquic (h) refinement of the FEM, and the one attained by the GFEM enrichment using bubble like functions, here referenced as selective (s) refinement. The convergence of the maximum deflection value of the given beam is shown in Figure 11.

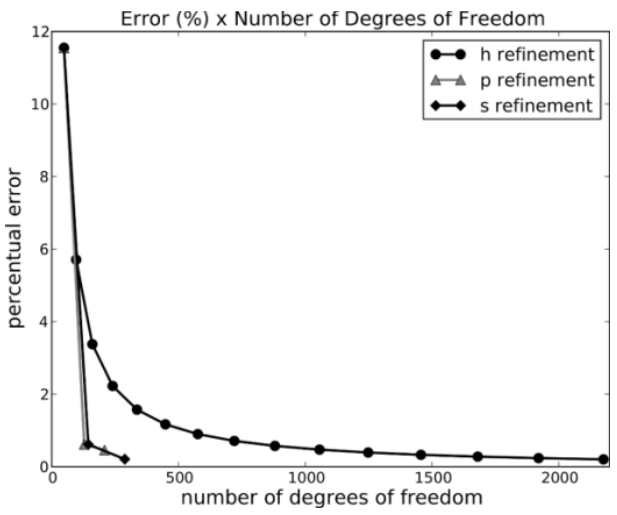


Figure 11 Convergence of the maximum deflection value for the hierarquic, polynomial and selective refinement.

The enrichment also enhances the quality of the stress field results. Figure 12 indicates the normal stress field in the horizontal and vertical directions, and the shear stress graphics distribution across the cross section positioned in the middle of the beam (x=*12.0*).

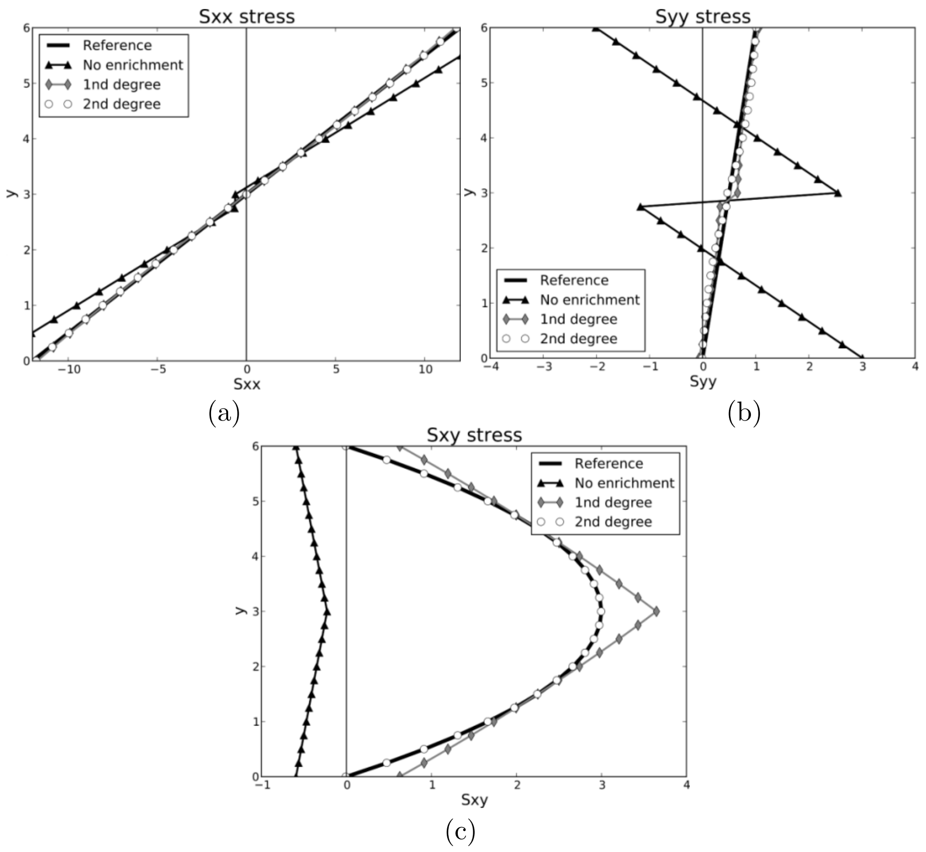


Figure 12 Normal horizontal (a), vertical (b) and shear (c) stress distribution across the cross section x = 12.0.

The same solid is finally used to test the code capability of treating different order triangular and quadrilateral elements to describe its geometry, all mixed in the same mesh. In order to do so, the non conventional mesh indicated in Figure 13 is then employed to solve the same problem.

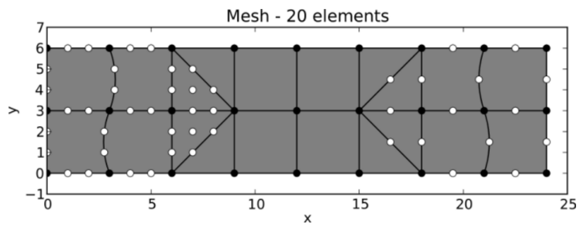


Figure 13 Non conventional mesh containing different order approximation triangular and quadrilateral elements.

The white nodes in Figure 13 are only used to define higher order geometries, i.e., present no degree of freedom associated, while the black ones are in fact nodes, and present conventional and enriched degrees of freedom associated, according to the enrichment applied. The horizontal normal stress results for a second order bubble like enrichment applied over it resulted in a field map close to the ones achieved before, by means of the regular mesh (Figure 14).

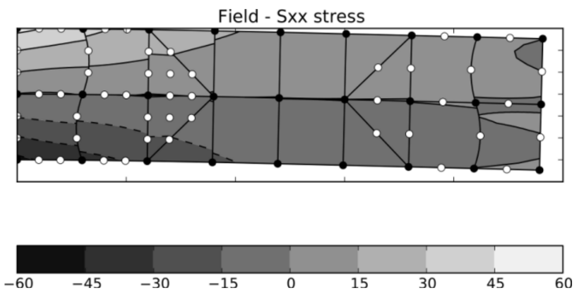


Figure 14 Horizontal normal stress field for a second order bubble like function enrichment achieved for the non conventional mesh

4.2 Two dimensional solid with a hole

The usefulness of the capability of using different order geometry description in the same mesh turns clear for problems containing circular shapes, as the one depicted in Figure 15.

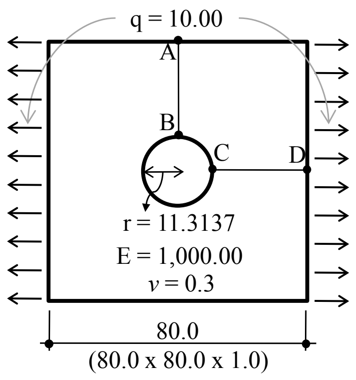


Figure 15 Solid with a hole (plane stress) model scheme.

The proposed mesh to solve it is a bilinear regular mesh, employing bi-cubic approximation for describing the geometry of the elements that surrounds the hole, as it is shown in Figure 16. As it can be observed, such elements present ‘white’ nodes in their facets, defining a cubic polynomial description for such elements’ sides.

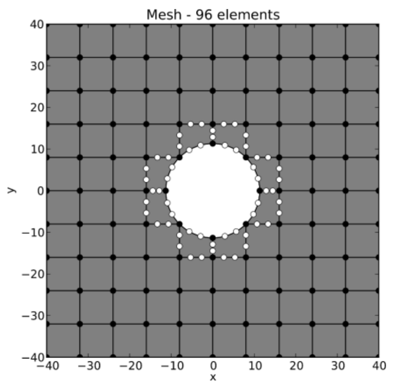


Figure 16 Bi-linear regular mesh used to describe the solid with a circular hole.

In order to attain good stress distribution results, the solids nodes are enriched using a full second degree enrichment (see equation (6)). The horizontal normal stresses in the solid, across the line A-B, and the vertical normal stress in the solid, across the line C-D, both indicated in Figure 15, are illustrated in Figure 17 and Figure 18, respectively.

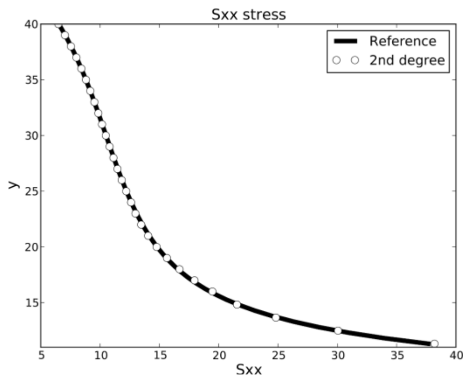


Figure 17 Horizontal normal stress in the solid, across the line A-B.

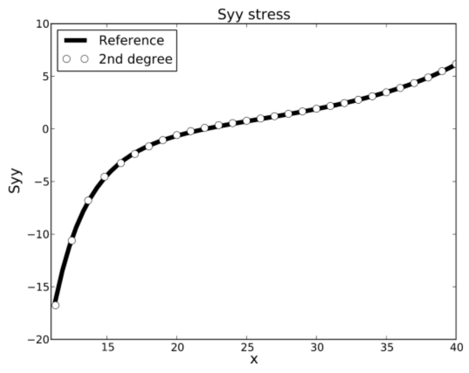
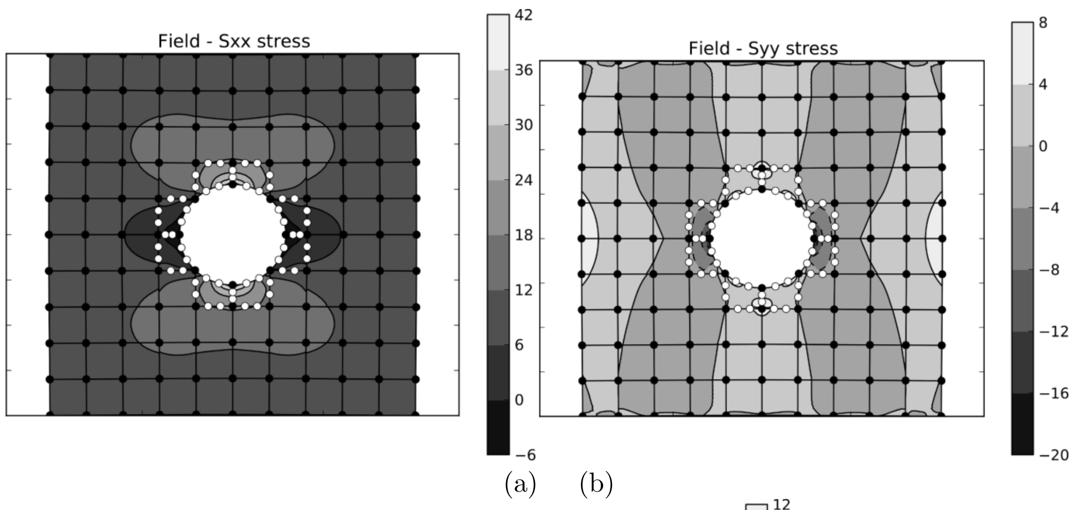


Figure 18 Vertical normal stress in the solid, across the line C-D.

Even though near the hole it is observed a great stress concentration, in both cases, the applied enrichment is sufficient to result in stresses values close to the reference solution. The reference solution results from a very fine mesh of second degree approximation triangular isoparametric elements, containing 13,662 elements and 27,637 nodes for a quarter of the solid. It is important to mention that the same region of the enriched solid results in circa 430 degrees of freedom, just a little fraction of the 55,274 degrees of freedom resulting from the discretization of the numerical reference solution.

The stresses field results are indicated in Figure 19 and are close to the ones of the reference numerical solution.



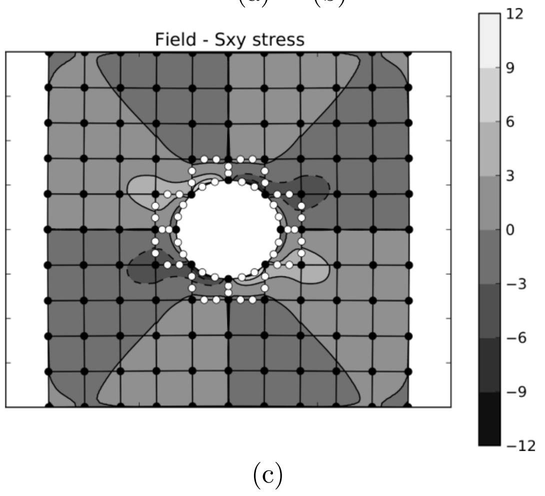


Figure 19 Normal horizontal (a), vertical (b) and shear (c) stress field maps.

4.3 Computational Performance

Once confirmed the code accuracy and flexibility to treat the geometry description using different approximation orders, the time performance turns out as another important aspect to be evaluated. As the computational effort depends basically on the total number of degrees of freedom involved, a problem with a simple geometry is hereby stated, as indicated in Figure 20.

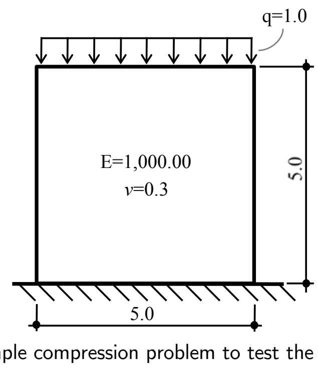


Figure 20 A simple compression problem to test the code performance.

The total processing time results are relative to a 2.6 version Python interpreter in an Intel Core i7 running at 2.80 GHz with 12GB RAM, on a GNU/Linux Ubuntu 10.04 64 bits Operating System. The achieved results in the proposed example are illustrated in Figure 21.

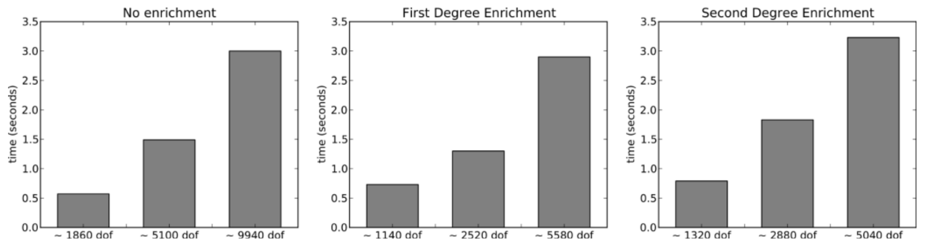


Figure 21 Computational performance (processing time, in seconds) for the different number of degrees of freedom and different polynomial enrichment degrees.

Even though the total processing time is a relative parameter, since it varies according to a variety of reasons, the main purpose of such evaluation is to demonstrate that the resulting processing time is suitable enough for the proposed application. It is important to notice that the number of degrees of freedom tested is much greater than the ones used in the previous stated examples, in which the results were accurate.

In addition to it, the GFEM requires relatively less degrees of freedom in order to achieve accurate results. For instance, to mention another case, the example addressed in section 4.2 demanded only 1,440 degrees of freedom in order to converge to the reference solution, which demands less than a second in order to solve it.

Moreover, analyzing the time reports generated by the program, by far the most time consuming task is the local stiffness matrix computation. This is due to the fact that each of its terms is computed in pure Python, which is not efficient for this kind of computation.

Actually, the results presented in Figure 21 can be optimized in several ways. For instance, the bottleneck tasks can be developed in a compiled programming language as C, once Python is efficient for calling and accessing other executable codes. Remarkable speed-ups are also related in Python's documentation, see Rossum(2011), using Python's Standard Library packages as, for instance, 'weave.inline', which allows inserting C codes inside the Python script. A good discussion about performance optimization can be found in Langtangen(2008).

Within this paper, we have chosen another alternative to illustrate Python's optimization potential. As each of the stiffness matrix computations is independent of the other elements, its parallelization is straightforward and demands changes only in few lines of the code. Therefore, the local stiffness matrix computations were performed in parallel, using a package of the Python's Standard Library called 'Multiprocessing', see Rossum(2011). It provides high level functions to easily support parallel programming. Considering that nowadays most of the desktop computers have multiple processing cores, this is an important programming tool that enables one to take full advantage of the used computer. A detailed description of this parallelization can be found in Piedade Neto et al(2011)

After these changes, one can process the same examples using more than one processing core. The results achieved for the same example stated in Figure 20, using four processing cores, are depicted in Figure 22.

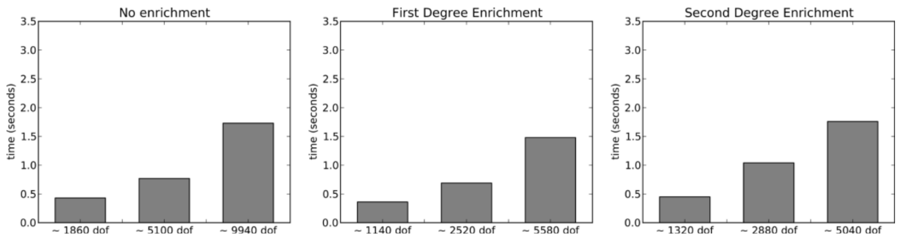


Figure 22 Processing time, in seconds – 4 processing cores, parallelization using Python's Multiprocessing package.

Finally, just to illustrate the potential for solving problem containing even larger number of degrees of freedom, Figure 23 indicates the total processing time to solve the same problem for meshes containing around 50,000 and 120,000 degrees of freedom, and first degree enrichment. The used computer is a Xeon X5660 running at 2.8 GHz and 48 GB RAM, containing 12 processing cores. In fact, problems up to 500,000 degrees of freedom were processed by means of the developed code.

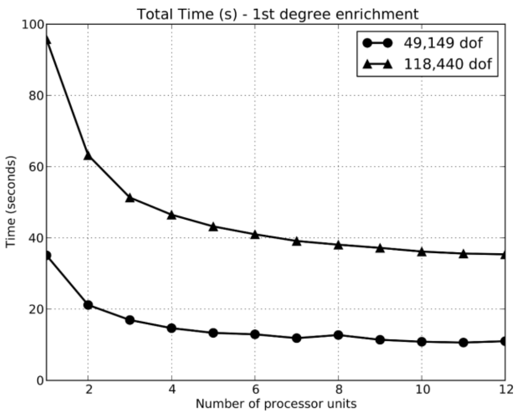


Figure 23 Total processing for meshes containing around 50,000 and 120,000 degrees of freedom, varying the number of processing cores from 1 to 12.

5 NONLINEAR SOLID MECHANICS ANALYSIS USING THE PROPOSED OO DESIGN

Analogously to the traditional FEM design presented in Figure 3, it is fully possible to extend the GFEM proposed OO design to add dynamic and nonlinear analysis features. It is important to mention that the nonlinear solid mechanics analysis using GFEM is a subject yet little discussed in the technical literature, reinforcing the importance of a GFEM OO design which allows one to solve such kind of problem. In order to do so for both cases, the first step is the inclusion of a class to control the time evolution. Just by including such class and adding some new methods to the 'element' and 'structural problem' classes, linear dynamic analysis can be performed using the same code framework proposed before. The use of the GFEM for dynamic analyses of trusses was already evaluated in Torri and Machado (2012).

Obviously, for nonlinear analysis, some numerical strategies for solving the nonlinear system of equation are necessary, as for instance the Newton-Raphson Method. Also, it is needed to include some new methods in the 'structural problem' class, but they are basically the same that would be necessary in a conventional FEM/OO code framework. In what follows, the main necessary changes in order to consider nonlinear solid mechanics are briefly commented.

- Nonlinear kinematics: it can be directly implemented in the proposed OO design, demanding only few additional methods to compute the stiffness matrices and load vectors in the deformed solid configuration. In fact, such methods were implemented in the element class in a short period of time, using a total Lagrangian description. An example of the Euler column is illustrated in Figure 24, showing both the deformed solid configuration and the numerical results of the relation between the axial force and the lateral deflection, depending on the amplitude of the initial imperfection 'e' in the mid span (*i.e.*, post-buckling equilibrium path).

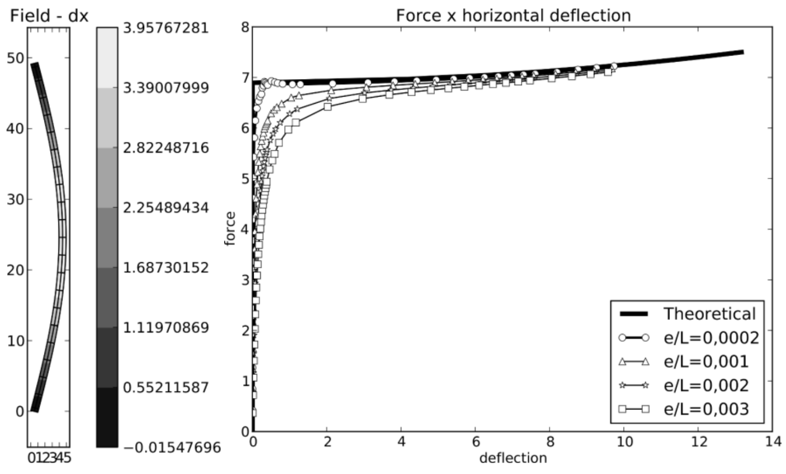


Figure 24 Euler column example relating the vertical force applied to the column to the maximum horizontal displacement, depending on the value of the initial imperfection 'e' (L is the column length).

- Elasto-plastic constitutive behavior: the material class requires methods to verify the hardening criterion, to compute the plastic strain evolution in time and the secant and tangent elasto-plastic constitutive relation. During the solution process, one also needs to store the plastic strain and hardening parameters state at the integration point. In order to do so, many different OO design solutions can be employed. For the current implementation of the proposed code a 'stress/strain state' class is defined, and instances of such class are then instantiated for each of the element's integration points, i.e., inside the element's instances. Such features were implemented in the code developed following the proposed OO design and were already validated using commercial FEM codes.

Figure 25 shows the nonlinear elasto-plastic behavior of a square shaped solid (plane strain hypothesis), illustrated by the nonlinear relation between the vertical displacementof a point located at the solid's top and the total force due a compression pressure applied over it.The depicted results are related to a nonlinear hardening plastic von Mises like material model. Linear polynomial enrichment was applied to the set of non constrained nodes.

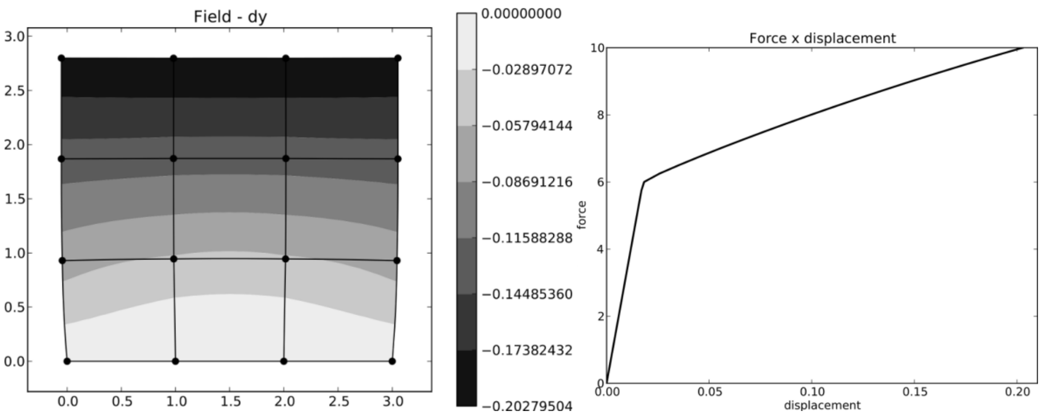


Figure 25 Nonlinear force x displacement relation for a nonlinear hardening von Mises material model and GFEM enriched square solid.

- Contact problems: it is necessary to create new classes to describe the contact elements and its targets. Such elements can be attached to the solid nodes or parts of its boundary, resulting in node-to-segment or segment-to-segment elements, for two-dimensional modeling. The targets, as the segment-to-segment contact elements, have its geometry and its physical behavior interpolation defined by means of one-dimensional PU, which already exists in the proposed OO design. In fact, the PU class instances of the proposed GFEM OO design were employed in order to implement segment-to-segment contact elements for frictionless contact problems, in the developed code. We find out that the generic proposed PU class promotes remarkable code reusability. Moreover, it naturally allows one to generate enriched contact elements, just following a similar strategy used for the enriched element's stiffness matrices. Even though such additional features demand major inclusions in the proposed OO class framework, it demands practically no changes in the overall proposed OO linear framework. Some illustrative examples of contact problems solved by means of it are shown in Figure 26.

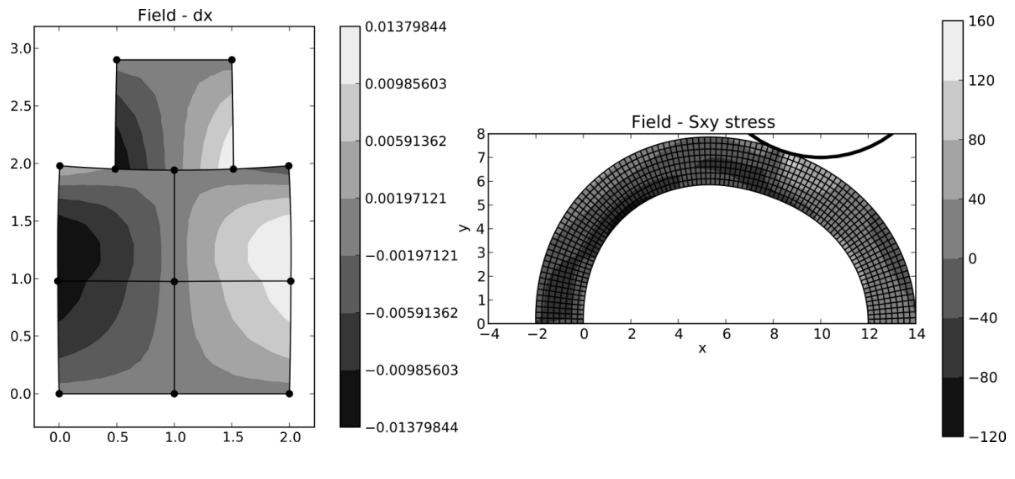


Figure 26 Contact problems solved using a nonlinear analysis code based on the proposed OO design.

6 CONCLUSIONS

Despite of its flexibility and power, the Generalized Finite Element Method demands an efficient computational implementation in order to take full advantage of such characteristics. In this paper we present an object-oriented class design in order to obtain a computational code which circumvents the difficulties due to the method's generality. Departing from a general object-oriented class design for the traditional Finite Element Method, a few new classes and issues are included in it in order to support the GFEM programming. Due to the methods characteristics, the use of Python programming language for the GFEM programming is technically justified. Some results computed using a code developed following such OO framework proves the efficiency of the propose design. Beyond the accuracy and flexibility achieved, the performance results prove that the advocated programming language present sufficient performance to be employed in numerical applications like this. Finally it is presented a brief discussion on how to expand the presented OO design in order to obtain a nonlinear analysis computational code.

References

Alves Filho, J. S. R., Devloo, P. R. B. (1991). Object Oriented programming in Scientific computations: the beginning of a new era. Engineering Computations, Vol. 8, Issue 1, pp. 81-87.

Bathe, K. J. (1996). Finite Element Procedures, Prentice-Hall, Inc. Englewood Cliffs, New Jersey.

Bittencourt, M. L. (2000).Using C++ templates to implement finite element classes, Engineering Computations. Vol. 17 No. 7, pp. 775-788.

Bordas, S. P. A., Nguyens, P. V., Dunant, C., Guidoum, A., Nguens-Dand, H. (2007). An extended finite element library, International Journal for Numerical Methods in Engineering, Vol. 71, pp. 703-732.

Cross, J. T., Masters, I., Lewis, R. W. (1999). Why you should consider object- oriented programming techniques for finite element methods. International Journal of Numerical Methods for Heat & Fluid Flow, Vol. 9 No. 3, 1999, pp. 333-347.

Dale, D., Droettboom, M., Firing, E. and Hunter, E. (2011).Matplotlib - Release 1.0.1.available at http://matplotlib.sf.net/Matplotlib.pdf, (acessed April 2011).

Duarte, C. A. and Oden, J. T. (1996a).An hpadaptative method using clouds. Computer Meth. in Applied Mechanics and Engineering, Vol 139, pp. 237-262.

Duarte, C. A. and Oden, J. T. (1996b).Hpclouds – an hpmeshless method. Numerical Methods for Partial Differential Equations, Vol. 12, pp. 673–705.

Dubois-Pélerin, Y., Zimmermann, T., Bomme, P. (1992).Object-oriented finite element programming: II A prototype program in Smalltalk. Computer Methods in Applied Mechanics and Engineering, Vol. 98, pp. 361-397.

Dubois-Pélerin, Y., Zimmermann, T. (1993).Object-oriented finite element programming: III. An efficient implementation in C++. Computer Methods in Applied Mechanics and Engineering, Vol. 108, pp. 165-183.

Forde, B. W. R., Foschi R. O., Stiemer, S. F. (1990). Object-Oriented Finite Element Analysis. Computer & Structures, Vol. 34, No. 3, pp. 355-374.

Hughes, T. J. R. (2000). The Finite Element Method – Linear Static and Dynamic Finite Element Analysis, Dover Publications, Inc.

Kuutii, J., Kolari, K. (2012). A local remeshing procedure to simulate crack propagation in quasi-brittle materials. Engineering Computations: International Journal for Computer-Aided Engineering and Software, Vol. 29 No. 2 pp. 125-143. Langtangen, H. P. (2008), Python scripting for computational science, Third Edition, Springer.

Layman, R. ,Missoum, S., Geest, J. V. (2010). Simulation and probabilistic failure prediction of grafts for aortic aneurysm. Engineering Computations: International Journal for Computer- Aided Engineering and Software, Vol. 27 No. 1, 84-105

Mackerle, J. (2000). Object-oriented techniques in FEM and BEM A bibliography (1996-1999). Finite Element in Analysis and Design, Vol. 36, pp. 189-196.

Mackie, R. I. (2000). An object-oriented approach to calculation control in finite element programs.Computer and Structures, Vol. 77, pp. 461-474.

Melenk, J.M. and Babuška, I. (1996). The Partition of Unity Finite Element Method: Basic Theory and Applications. Seminars fur AngewandteMathematik, EidgenossischeTechnisheHochschule, Research Report No. 96-01, January, CH-8092 Zurich, Switzerland.

Pereira, J. P. A. (2004). Extração de fatores de intensidade de tensão utilizando a solução do Método dos Elementos Finitos Generalizados.Master of Science Dissertation, São Carlos Engineering School, University of São Paulo.

Piedade Neto, D. ; Ferreira, M. D. C. ; Proença, S. P. B.(2011). Generalized Finite Element Method Computation: parallelization using Python multiprocessing package. XIX Congreso sobre Métodos Numéricos y sus Aplicaciones, 2011, Rosário. Mecánica Computacional. Rosário : Associación Argentina de Mecánica Computacional, 2011. v. 30. p. 3045-3061.

Rossum, G. V. (2011). The Python Library Reference - Release 2.7.2, 2011.available at http://docs.python.org/download.html. (acessed July 2011).

Tucker, A. B. and Noonan, R. E. (2008). Linguagens de programação – princípios e paradigmas. SegundaEdição, McGraw Hill.

The Scipy Community (2010). NumPy Reference – NumPy v1.5 Manual (DRAFT). available at http://docs.scipy.org/doc/numpy-1.5.x/reference/, (acessed 08 April 2011).

The Scipy Community (2011). SciPy - SciPy v0.9 Reference Guide (DRAFT). available at http://numpy.scipy.org/, (acessed 08 April 2011).

Torri, A. J., Machado, R. D. (2012). Structural dynamic analysis of time response of bars and trusses using the generalized finite element method.Latin American Journal of Solid and Structures. Vol. 9, pp 309 -337.Zimmermann,

T., Dubois-Pélerin, Y., Bomme, P. (1992). Object-oriented finite element programming: I. Governing principles. Computer Methods in Applied Mechanics and Engineering, Vol. 98, pp. 291-303.