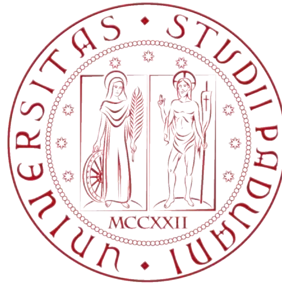

UNIVERSITÀ DEGLI STUDI DI PADOVA
Dipartimento di Ingegneria Industriale DII
Corso di Laurea Magistrale in Ingegneria dei Materiali

Acceleration of FEM Procedures in Python: Application to the RVE Analysis of Composite Materials

Relatore
ch.mo Prof. Angelo Simone

Laureando
Pietro Fumiani
matr: 1157334

Anno Accademico 2019/2020

Abstract

In this thesis project, an originally very simple Finite Element Method (FEM) code written in Python is accelerated using well-known techniques: sparse format for matrices and vectorization of operations. The estimate of the transverse elastic modulus of unidirectional fiber-reinforced composite is used as a real-world testing situation for the FEM package called `feat` [9]. For this purpose, a simple Representative Volume Element (RVE) analysis is created. The performance of the FEM code is very satisfying, the code being able to solve a 2D problem in linear elasticity with about 1 million degrees of freedom in around 30 seconds. Regarding the transverse modulus estimate, taking into account all the simplifications employed in the modeling, we can consider the result as reasonable. This project shows that a very good computational performance can be achieved even with a simple didactic code as a starting point.

Riassunto

In questo lavoro di tesi, un codice FEM inizialmente molto semplice scritto in Python viene accelerato usando tecniche note: memorizzazione in formato sparso per le matrici e vettorizzazione delle operazioni. La stima del modulo elastico trasversale di un composito rinforzato con fibre unidirezionali viene usata come situazione reale di test per il pacchetto FEM che prende il nome di `feat` [9]. Per questo scopo viene realizzata una semplice analisi del Representative Volume Element (RVE). La prestazione ottenuta dal codice FEM è molto soddisfacente (un problema 2D di elasticità lineare con 1 milione di gradi di libertà viene risolto in circa 30 secondi). Per quanto riguarda la stima del modulo trasversale, tenendo presente tutte le semplificazioni utilizzate nella modellazione, possiamo considerare i risultati ottenuti come ragionevoli. Questo progetto mostra che si possono raggiungere prestazioni molto buone anche partendo da un codice scritto in un contesto didattico.

Contents

Preface iv

1 Micromechanical Estimates 1
- 1.1 Transverse Modulus 1
- 1.2 Analytical Micromechanical Models 1
  - 1.2.1 Reuss Model 2
  - 1.2.2 Halpin-Tsai Equation 3

2 Numerical and Coding Tools 5
- 2.1 Finite Element Method 5
- 2.2 RVE Analysis 8
- 2.3 Python 13
- 2.4 Gmsh 14
- 2.5 Hardware 14

3 Acceleration 15
- 3.1 Base Implementation 15
- 3.2 Sparse Format for Matrices 17
- 3.3 Vectorized Assembly 21

4 Results and discussion 25
- 4.1 FEM Acceleration 25
- 4.2 RVE Analysis Results 28
- 4.3 Conclusions 31

Bibliography 33

Preface

This project was born with a mainly educational objective. During the Computational Mechanics of Materials course a very simple finite element code has been developed. It was the first time for me writing code to complete a certain task. By the end of the course I realized that I got interested in the study of Finite Element Method and in programming. With the supervisor, we decided that work on FEM coding for my master thesis project could be a good chance to develop my newly discovered interest. The idea is to start from the code developed during the course and try to accelerate its performance. The plan is to find out how far a code indipendently written by a student can go. Sometimes, Finite Element commercial software packages are considered as irreplaceable tools with super powers. They are surely reliable instruments for a large number of tasks in different branches of engineering, but with this work we hope to point out that it is possible to obtain good performance also from code developed by a student. To test the developed package (`feat` [9]) on a real task we decided to focus our attention on the estimation of the elastic properties of composite materials. A procedure to get an estimate of the elastic modulus has been implemented. The present methodology uses some simplifications to make implementation easier. For this reason we want to point out that the accuracy of the obtained estimate is not the goal of this work. The main purpose of this work is to develop a deeper knowledge of FEM by trying to improve the performance of our program.

Pietro Fumiani
Venezia
July 2020

Chapter 1

Micromechanical Estimates

1.1 Transverse Modulus

The determination of elastic moduli is a critical task for the mechanics of composite materials. Stiffness, together with strength, is a fundamental factor in structural design procedures [18]. Moreover the estimate of the different moduli that characterize a composite material is essential for material design too. In other words, the capability of computing moduli is useful to understand how to achieve the desired mechanical properties combining different materials in the best proportion and using the best spatial structure.
This work focuses on the determination of the transverse Young's modulus of unidirectional fiber-reinforced composite. A schematic representation of the material (Figure 1.1) is used in order to define the transverse modulus. Fibers are embedded in the matrix and aligned in direction 1. The transverse modulus is the apparent Young's modulus of the material in the direction transverse to the fibers, i.e., direction 2. The symbol used to indicate this modulus is E_2.
Usually the evaluation of effective elastic properties of composite is handled using the micromechanics of composite materials. In the context of this theory the transverse modulus is defined by Jones [18] as a *matrix-dominated* property. This term is used to underline the fact that the matrix modulus can influence the value of E_2 much more than the fiber modulus. The following section gives a simple definition of micromechanics of composite and its primary tools: analytical models.

1.2 Analytical Micromechanical Models

As Stated by Jones [18]: "Micromechanics — *The study of composite material behavior wherein the* interaction *of the constituent materials is examined in detail as part of the definition of the behavior of the* heterogeneous *composite material.*". Micromechanics employs a variety of different analytical models that are mainly based on two different theoretical methods:

- mechanics of material;
- elasticity.

The main purpose of these analytical models is the prediction of the composite lamina elastic moduli. The estimate is based on the constituents moduli, materials arrange-

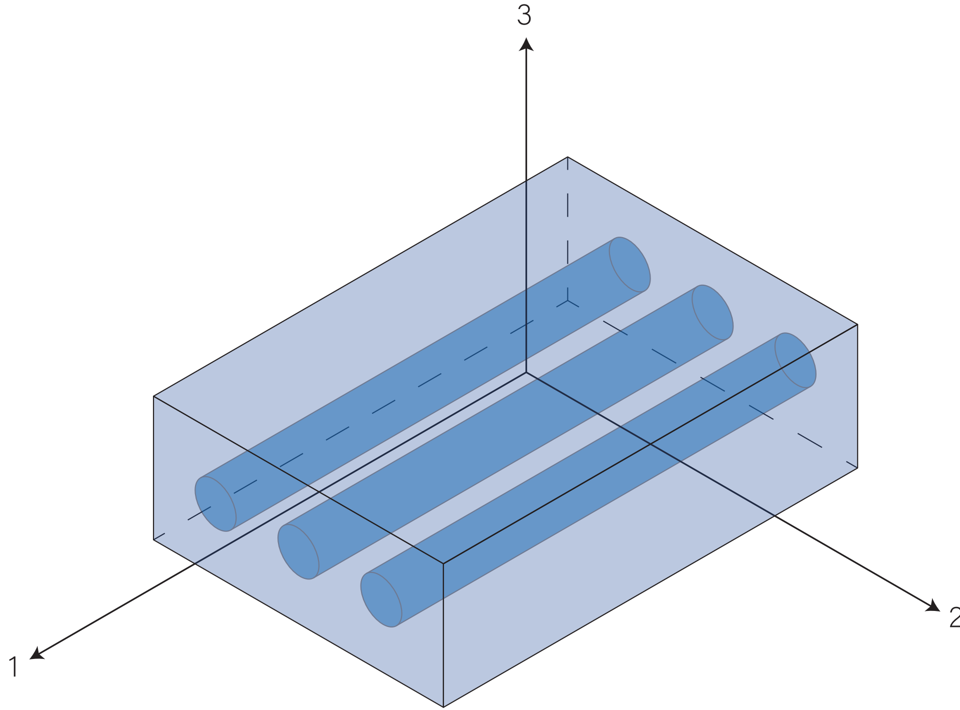

Figure 1.1: Unidirectional fiber-reinforced composite with principal coordinate system.

ment in space, and relative proportions between constituents. Moduli resulting from these models need to be validated through a comparison with elasticity properties obtained from experimental measures. Thereafter, the real benefit of these mathematical tools can be really appreciated. Indeed these models make it possible to define a rough estimate of the elastic modulus without the need to manufacture the material and test it.

Although these models are widely used, around their employment there has always been a certain controversy as to which of the models was the most suitable for different situations. This debate is also caused by the fact that the accuracy of the micromechanical models is quite low for properties controlled by matrix elasticity quantities. In this sense, micromechanics analysis should be considered mainly as providers of qualitative information rather than as quantitative data. Micromechanics is surely a guide for composite materials design but it is in general incorrect to think that an analytical model can return exact quantitative prediction about material performance [18, 12]. In the rest of this section an overview of the main analytical models is given focusing not only on features but also on drawbacks. The following description deals only with models for the prediction of the transverse elastic modulus (E_2).

1.2.1 Reuss Model

The first and simplest model for transverse modulus determination is the Reuss model or inverse rule of mixtures, this model is based on the mechanics of material approach. The modulus computed with this model is also known as the lower-bound modulus because Paul in 1960 showed that the inverse rule of mixtures is actually the lower

bound on E_2 [12]. The expression to compute the transverse modulus is:

$$E_2 = \frac{E_f E_m}{V_m E_f + V_f E_m}. \tag{1.1}$$

The fundamental assumption in the development of this model is that both fibers and matrix are subject to the same stress when loaded in the direction normal to the fibers. Even though the Reuss model is very simple to use, its main hypothesis leads to a large difference between its estimate, numerical results, and experimental results. Jones [18] underlines that the equality between stress in the fiber and stress in the matrix is not realistic because the two Poisson's ratios are usually different. Additionally the difference in Poisson's ratios determine also che presence of longitudinal stresses in both constituents combined with shearing stresses at the boundary between the different materials. For these reasons the model clearly underestimates the value of E_2 compared with experimental values. This inaccuracy has been confirmed also by recent studies such as that of Goudarzi and Simone [13]. They compared transverse modulus values obtained with a FEM analysis and the estimate given by the inverse rule of mixtures. The model is generally in bad agreement with the more accurate computational result and the error grows as the matrix Poisson's ratio approach 0.5, i.e., towards the incompressibility limit of the matrix material.

1.2.2 Halpin-Tsai Equation

Halpin and Tsai [15] developed a simple approximate relation starting from the more complicated Herrmanns' micromechanical solution [16]. The equation for the evaluation of the transverse modulus is given by

$$\frac{E_2}{E_m} = \frac{1 + \xi \eta V_f}{1 - \eta V_f}, \tag{1.2}$$

with the parameter

$$\eta = \frac{(E_f/E_m) - 1}{(E_f/E_m) + \xi}. \tag{1.3}$$

The quantity indicated with ξ is a curve fitting parameter and can be considered as a measure of the reinforcement given by fibers to the material. The value to use for ξ in the particular case of transverse modulus is usually $\xi = 2$. The reason for this is that the two authors observed a very good agreement with the finite difference method solution developed by Adams and Doner [1, 2]. This solution, used as a reference for the interpolation, is related to the evaluation of the transverse modulus for a square array of circular fibers with a particular fiber volume fraction of 0.55. Halpin and Tsai tested their work also against results obtained by Foye [8] for a diamond array of rectangular fibers. They found out that to have an excellent fit with Foye's solution it was necessary to change the value of ξ from 2 to

$$\xi = 2\frac{a}{b}, \tag{1.4}$$

where a/b is the rectangular cross-section aspect ratio. This difference in the value of ξ means that the estimate of transverse modulus given by Halpin-Tsai equation changes according to the geometry of the cross-section (circular or rectangular) and the spatial arrangement (square array or diamond array) of the fibers. According to Jones[18], the fact that the predictions made by the Halpin-Tsai equation vary for different arrays

implies that it is not possible to obtain a precise measure of the transverse modulus using (1.2). Both fiber arrays previously described and used for fitting are characterized by a periodic structure. This kind of spatial arrangement is nearly impossible to observe in a real-world material. Observing the structure of fiber reinforced specimens (e.g., see [12, 26]) the random nature of fiber-packing geometry is evident. Although the Halpin-Tsai equations has been widely and successfully used in composite materials design for a long time, it has to be considered as an approximation.

Chapter 2

Numerical and Coding Tools

The main task of this work is to create a fast computational framework. As an application, the framework is employed to provide a rough estimate of E_2 for unidirectional fiber-reinforced composite materials. In the previous section it has been shown that commonly used micromechanical models contain somehow a certain approximation that causes inaccuracy. The common alternative to analytical instruments is the use of numerical methods like finite difference and finite element. The instruments adopted in this work are described in the following sections and can be divided into two categories:

- theoretical tools:
  - Finite Element method (FEM);
  - Representative Volume Element (RVE) Analysis;
- operative tools:
  - Python programming language;
  - Gmsh finite element mesh generator.

2.1 Finite Element Method

Only a simple summary of necessary informations of the Finite Element Method is provided. For a comprehensive account on this widely adopted and powerful tool see [3].
FEM is used to get an approximate solution for field problems described by differential equations. In linear elasticity problems the primary field is the displacement field and the differential equation is the equilibrium equation. In this work we deal with a plane strain problem which involves the following equations:

$$\begin{bmatrix} e_{xx} \\ e_{yy} \\ 2e_{xy} \end{bmatrix} = \begin{bmatrix} \partial/\partial x & 0 \\ 0 & \partial/\partial y \\ \partial/\partial y & \partial/\partial x \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix}, \qquad \text{strain-displacement equation} \quad (2.1)$$

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = \begin{bmatrix} E_{11} & E_{12} & E_{13} \\ E_{12} & E_{22} & E_{23} \\ E_{13} & E_{23} & E_{33} \end{bmatrix} \begin{bmatrix} e_{xx} \\ e_{yy} \\ 2e_{xy} \end{bmatrix}, \qquad \text{constitutive equation} \quad (2.2)$$

$$\begin{bmatrix} \partial/\partial x & 0 & \partial/\partial y \\ 0 & \partial/\partial y & \partial/\partial x \end{bmatrix} \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \qquad \text{equilibrium equation} \quad (2.3)$$

In Equation (2.3), b_x and b_y are the component of the body force vector. The matrix of elastic constants in Equation (2.2) for plane strain is given by:

$$\mathbf{E} = \begin{bmatrix} E_{11} & E_{12} & E_{13} \\ E_{12} & E_{22} & E_{23} \\ E_{13} & E_{23} & E_{33} \end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ 1 & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix}. \tag{2.4}$$

Equations (2.1)–(2.3) can be expressed in a more compat fashion using their matrix equivalent form as follows:

$$\mathbf{e} = \mathbf{D}\mathbf{u}, \qquad \boldsymbol{\sigma} = \mathbf{E}\mathbf{e}, \qquad \mathbf{D}^T\boldsymbol{\sigma} + \mathbf{b} = \mathbf{0}. \tag{2.5}$$

The variational formulation is based on the Minimum Potential Energy principle. The total potential energy (TPE) of a body is given by

$$\Pi = U - W, \tag{2.6}$$

where U is the internal energy and is represented by the elastic strain energy

$$U = \frac{1}{2}\int_\Omega h\boldsymbol{\sigma}^T\mathbf{e}\,\mathrm{d}\Omega = \frac{1}{2}\int_\Omega h\mathbf{e}^T\mathbf{E}\mathbf{e}\,\mathrm{d}\Omega, \tag{2.7}$$

where h is the thickness of the body. Note that the second form in the equation is obtained inserting Equation (2.2) into che first form. The term W indicates the external energy that includes the contributions from interior (body) and boundary forces:

$$W = \int_\Omega h\mathbf{u}^T\mathbf{b}\mathrm{d}\Omega + \int_{\Gamma_t} h\mathbf{u}^T\hat{\mathbf{t}}\mathrm{d}\Gamma. \tag{2.8}$$

In these equations, Ω and Γ are respectivley the plane domain and its boundary. In particular, Γ_t is the portion of the boundary where traction forces $\hat{\mathbf{t}}$ are applied. Using the Minimun Potential Energy principle the fact that the solution $u^*(x)$ satisfies the governing equations results in the potential energy Π being stationary:

$$\delta\Pi = \delta U - \delta W = 0 \tag{2.9}$$

with respect to variations $u = u^* + \delta u$ of the exact displacement $u^*(x)$. Now applying the finite element discretization of the domain it is possible to decompose the TPE functional into a sum of terms each related to one element of the discretized domain. The same operation is applied also to the stationary condition (2.9). Then, for a certain element e we have:

$$\delta\Pi^e = \delta U^e - \delta W^e = 0. \tag{2.10}$$

The finite element method is based on the fact that the domain is discretized into a set of element that create a mesh that represent the domain geometrically. If we consider a generic element the method involves replacing the exact displacement with an approximation:

$$u^*(x) \approx u^e(x). \tag{2.11}$$

The approximate displacement field obtained by the interpolation of the nodal displacement and it is expressed as follows:

$$u_x(x,y) = \sum_{i=1}^{n} N_i^e(x,y)\,u_{xi}\,, \qquad u_y(x,y) = \sum_{i=1}^{n} N_i^e(x,y)\,u_{yi}\,. \tag{2.12}$$

$N_i^e(x,y)$ are the element shape functions. Each shape function must have certain properties: it has to be continuous, it has a value of one at the node it is associated with and zero at all other nodes. The last equation is represented in a more convenient way using matrix form:

$$\mathbf{u}(x,y) = \begin{bmatrix} u_x(x,y) \\ u_y(x,y) \end{bmatrix} = \begin{bmatrix} N_1^e & 0 & N_2^e & 0 & \cdots & N_n^e & 0 \\ 0 & N_1^e & 0 & N_2^e & \cdots & 0 & N_n^e \end{bmatrix} \mathbf{u}^e = \mathbf{N}\mathbf{u}^e\,. \tag{2.13}$$

From the finite element displacement field the strain field is also obtained as:

$$\mathbf{e}(x,y) = \begin{bmatrix} \frac{\partial N_1^e}{\partial x} & 0 & \frac{\partial N_2^e}{\partial x} & 0 & \cdots & \frac{\partial N_n^e}{\partial x} & 0 \\ 0 & \frac{\partial N_1^e}{\partial y} & 0 & \frac{\partial N_2^e}{\partial y} & \cdots & \frac{\partial N_n^e}{\partial y} & \\ \frac{\partial N_1^e}{\partial y} & \frac{\partial N_1^e}{\partial x} & \frac{\partial N_2^e}{\partial y} & \frac{\partial N_2^e}{\partial x} & \cdots & \frac{\partial N_n^e}{\partial y} & \frac{\partial N_n^e}{\partial x} \end{bmatrix} \mathbf{u}^e = \mathbf{B}\,\mathbf{u}^e\,, \tag{2.14}$$

where **B** is called the strain-displacement matrix and is defined as **B** = **DN**. If we insert relations (2.13), (2.14) and the matrix form of (2.2) into equations (2.8) and (2.7) for the generic element we obtain:

$$U^e = \frac{1}{2}\int_{\Omega^e} h\,\mathbf{e}^T\mathbf{E}\mathbf{e}\,\mathrm{d}\Omega^e = \frac{1}{2}\int_{\Omega^e} h\,\mathbf{u}^T\mathbf{B}^T\mathbf{E}\mathbf{B}\mathbf{u}\,\mathrm{d}\Omega^e\,, \tag{2.15}$$

$$W^e = \int_{\Omega^e} h\,\mathbf{u}^T\mathbf{N}^T\mathbf{b}\mathrm{d}\Omega^e + \int_{\Gamma^e} h\,\mathbf{u}^T\mathbf{N}^T\hat{\mathbf{t}}\mathrm{d}\Gamma^e\,. \tag{2.16}$$

We define two quantities that are fundamental for the final form of the problem: the element stiffness matrix:

$$\mathbf{K}^e = \int_{\Omega^e} h\mathbf{B}^T\mathbf{E}\mathbf{B}\,\mathrm{d}\Omega^e\,, \tag{2.17}$$

and the nodal force vector:

$$\mathbf{f}^e = \int_{\Omega^e} h\mathbf{N}^T\mathbf{b}\,\mathrm{d}\Omega^e + \int_{\Gamma^e} h\mathbf{N}^T\hat{\mathbf{t}}\,\mathrm{d}\Gamma^e\,. \tag{2.18}$$

Then the TPE becomes:

$$\Pi^e = U - W = \frac{1}{2}\mathbf{u}^{eT}\mathbf{K}^e\mathbf{u}^e - \mathbf{u}^{eT}\mathbf{f}^e \tag{2.19}$$

and if we take the variation of the preeceding expression with respect to the displacements the resulting expression is

$$\delta\Pi^e = (\delta\mathbf{u}^e)^T\frac{\partial\Pi^e}{\partial\mathbf{u}^e} = (\delta\mathbf{u}^e)^T[\mathbf{K}^e\mathbf{u}^e - \mathbf{f}^e] = 0\,. \tag{2.20}$$

The variation $\mathbf{u}^e$ is arbitrary so the term inside brakets mush be zero, therefore

$$\boxed{\mathbf{K}^e\mathbf{u}^e = \mathbf{f}^e} \tag{2.21}$$

This expression represent the discrete version of the equilibrium equation that needs to be assembled into the global system together with contributions from every other element in the mesh. The assembly process is resposible for the construction of the global stiffness equation that is later solved to obtain the solution: the vector of nodal displacements **u**. This is done using global node numbering to map element degrees of freedom with global degrees of freedom (see Section 3.1). After the assembly of the global stiffness matrix the system is ready for the application of the boundary conditions. In the end, the the global system of equations, represented in matrix format by

$$\boxed{\mathbf{K}\mathbf{u} = \mathbf{f},} \tag{2.22}$$

can be solved for the displacement vector. For the present work the solution vector is the starting point for the computation of the transverse modulus of the composite material domain. The employed method is described in the follwing section.

2.2 RVE Analysis

In this work, a procedure based on the concept of Representative Volume Element (RVE) is used. Hill probably gave the first formal definition of RVE in 1963 [17] saying that the RVE :

- "is structurally entirely typical of the whole mixture on average",
- "contains a sufficient number of inclusions for the apparent overall moduli to be effectively independent of the surface values of traction and displacement...".

Drugan and Willis [6] give another interesting definition of this term: "It is the smallest material volume element of the composite for which the usual spatially constant (overall modulus) macroscopic constitutive representation is a sufficiently accurate model to represent mean constitutive response."

The approach developed in this project is based on the work of Terada et al. [26]. They proposed to analyze a series of simulations carried out on domains (RVEs) of increasing size. The key concept here is that if the RVE is too small the positioning of fibers inside the domain is still able to influence the obtained elastic properties. The size of the domain is not sufficient to make its own elastic properties independent from that particular configuration of the fibers in matrix. Namely, the domain is not an RVE, it is not representative of the overall material. In their paper, Terada et al. [26] studied the convergence of the norm of the elasticity matrix (stress-strain matrix). This matrix is calculated using a homogenization procedure that is not employed in this work. The resuts of their work are well summarized by Figure 2.1. In the graph three different curves have been plotted, each representing the result obtained applying different boundary conditions to the same domain. This is useful to us because it proves that it is possible to reach the same result with all types of boundary conditions.

Here, the focus is on the computation of the transverse modulus of unidirectional fiber-reinforced composite materials and, because of the parallelism of fibers, a two-dimensional model is used to describe the material. The domain is a square in which fibers are modeled as disks and the remaining surface is considered as matrix material. As verified by Terada et al. [26] and Goudarzi and Simone [13], the best approach to estimate elastic constants (e.g. E_2) is the application of periodic boundary conditions

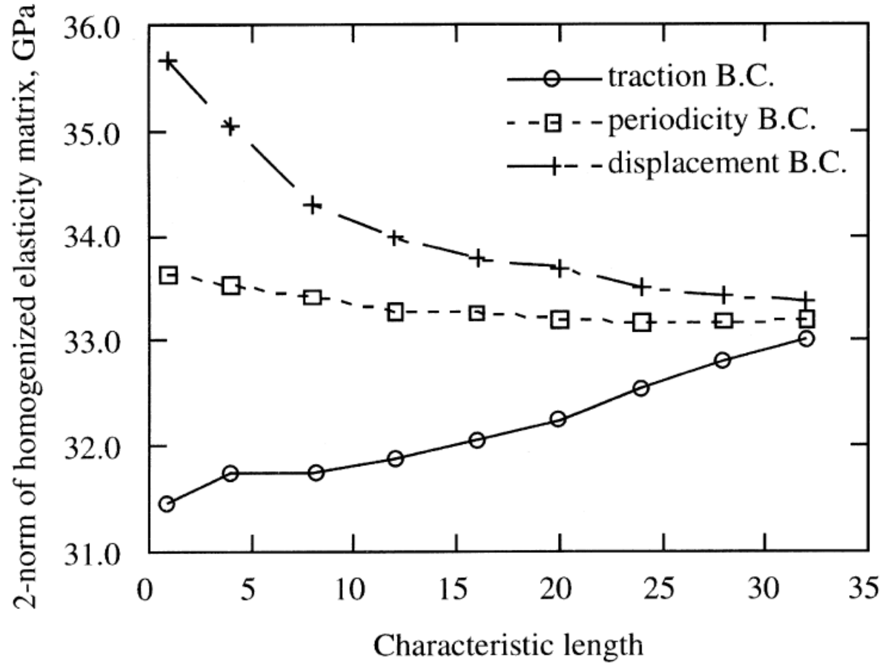

Figure 2.1: Convergence associated with orthotropy and isotropy for homogenized elasticity matrix. (from Terada et al. [26])

within a periodic mesh. One of the reasons because periodic conditions are preferred is clear in Figure 2.1. The convergence of effective elastic properties is faster in the case of periodic boundary conditions respect to the case of displacement or traction conditions. That is to say, convergence is reached for smaller RVE, which in turn represent an easier problem to be solved from the computational point of view. Nevertheless the implementation of periodic boundary conditions is considered an unnecessary task for the purpose of this thesis. Hence the estimate of the transverse modulus is performed using a simplified procedure based on that used by Dong [5]. His method make use of the periodic boundary conditions but here we employ normal Dirichlet (essential) boundary conditions. A graphical representation of the geometry of a generic RVE domain is given in Figure 2.2. The bottom left corner of the RVE is locked, the rest of the left side is not allowed to move in direction 2. The mechanical load is applied on the model through an imposed constant displacement $\bar{u}$ over the whole right-hand side of the square.

Now let's see how the transverse modulus is computed for a generic RVE. The imposition of homogeneus Dirichlet boundary conditions ($\bar{u} = 0$) generates support reactions for all degrees of freedom (DOFs) subject to that conditions. Using global sitffness matrix $\mathbf{K}$ rows relative to that constrained DOFs along with the displacements vector, it is possible to compute reaction forces for the above mentioned degrees of freedom. The useful data from $\mathbf{K}$ are lost during boundary conditions imposition, therefore they must be saved before this operation. In this way it is possible to compute the reaction forces using the following matrix relation:

$$\mathbf{f}^* = \mathbf{K}_\mathrm{R}\mathbf{u} \tag{2.23}$$

where $\mathbf{K}_\mathrm{R}$ indicate the rows of the stiffness matrix related to degrees of freedom for

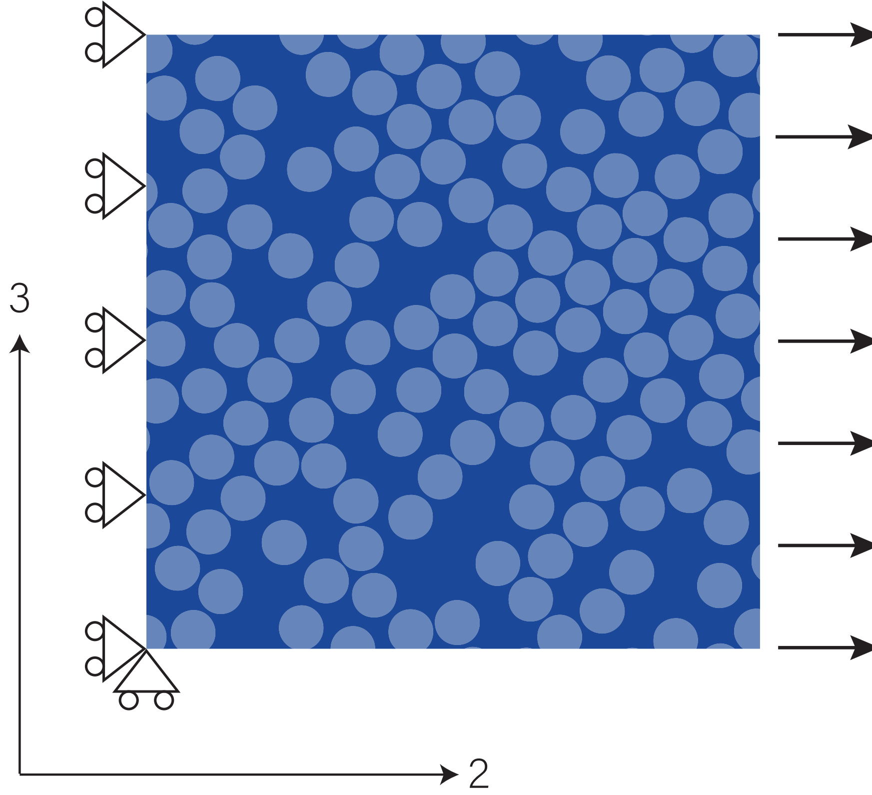

Figure 2.2: RVE domain geometry and boundary conditions.

which $\bar{u} = 0$, that is to say degrees of freedom that present a reaction force. Then, assuming for the domain a condition of plane strain, the stress in the direction transverse to the fibers can be evaluated as:

$$\sigma_2 = \frac{R}{A} = \frac{R}{Lh} \tag{2.24}$$

where

$$R = \sum_{i=1}^{m} f_i^* \tag{2.25}$$

is the sum of all contributions of nodal reaction forces, L is the side lenght of the RVE and h is the thickness of the domain. The strain can be calculated as:

$$\varepsilon_2 = \frac{L_f - L_i}{L} = \frac{\bar{u}}{L} \tag{2.26}$$

using the imposed displacement value $\bar{u}$ and the side of the RVE. Finally a rough estimate of the transverse modulus of the RVE is given by the constitutive equation 2.2:

$$E_2 = \frac{\sigma_2}{\varepsilon_2}. \tag{2.27}$$

These steps to obtain an estimate of E_2 are performed in the post-processing phase of each FEM simulation because the displacements vector **u**, which is unknown until the final solution of the main FEM system, is the starting point. To ensure that the approximate solution obtained with FEM is accurate, a mesh refinement study is essential. This operation, here done through h-refinement, consists of a certain number

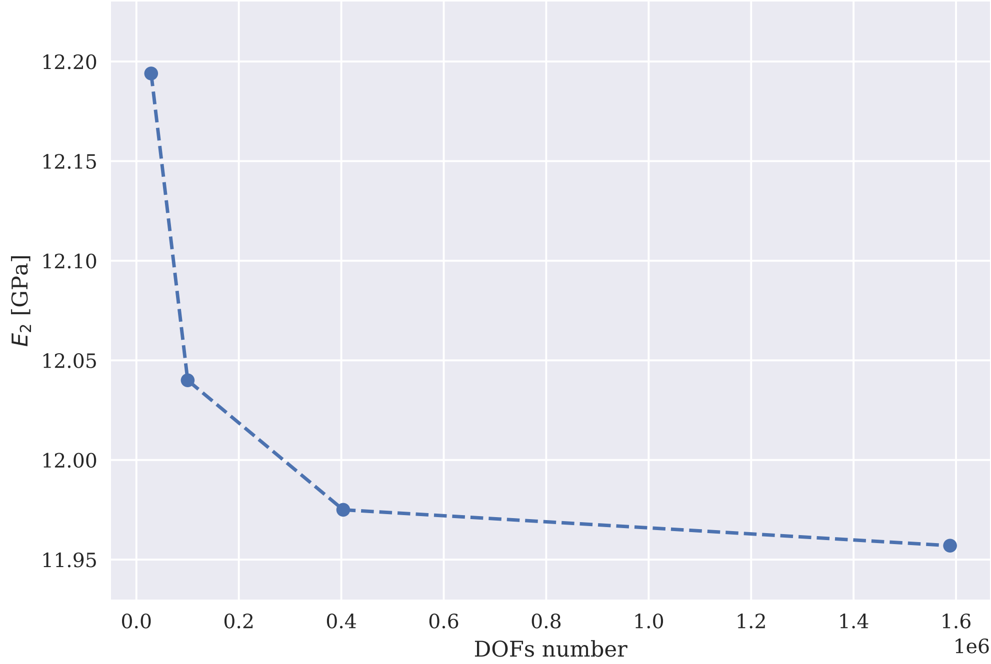

Figure 2.3: Example of mesh refinement procedure that leads to convergence.

of subsequent finite element analyses in which the element size is gradually reduced. Therefore, increasing the total number of elements in the mesh by downsizing the single element, allows the analysis result to get closer to the exact solution of the problem. As depicted in Figure 2.3 at some point a threshold is reached and any further mesh refinement takes to the same numerical results. Now that the process to obtain E_2 for a given RVE has been explained, we procede with the description of the RVE Analysis. The key concept used for the analysis are the “sample” and the “step”. A sample consists of a sequence of domains called steps. Domains that belongs to a sample are all created starting from the so called “main step”. All the other steps (domains) are obtained by “cutting” a square of the desired size from the main step (see Figure 2.5). This cut operation is performed so that all steps and the main step share the same center. We can say that steps are a set of “concentric” squares evenly spaced and sorted by increasing side lenght. The input data for an RVE analysis are: the fibers volume fraction Vf, the radius of the fibers R and the side lenght of the main step L. The arrangement of the fibres must be random to accurately reproduce the microstructure of the material. For this reason, each sample is associated to a particular numerical seed that is used to initialize a random number generator. The generator is employed by the domain creation algorithm. The most common and simple algorithm to create randomly distributed fibers is the Random Sequential Absortion (RSA) followed for example by Kari [19]. Unfortunately the maximum volume fraction that is possible to obtain using RSA is about 54.7% [29, 27]. This is the saturation that is practically impossible to reach in a reasonable number of iterations. The RSA algorithm has been implemented but it has confirmed to be realiable for V_f until 40–45%. The majority of experimental data found in literature are related to materials with fibers volume fractions from 50% to 60%. It is important to have the possibility to compare the numerical results of

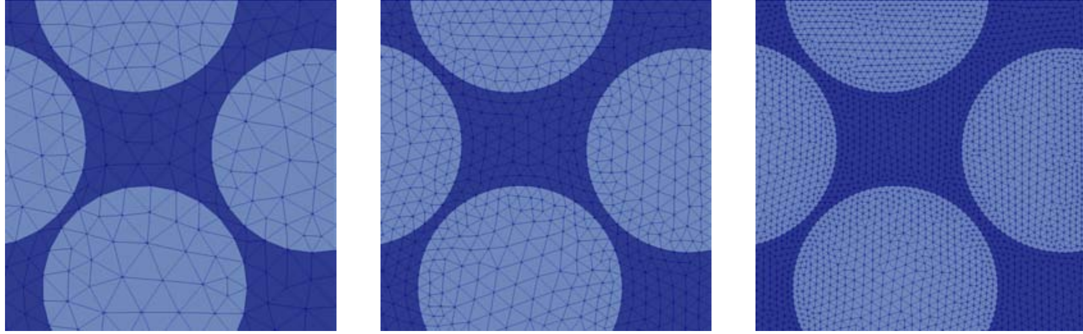

Figure 2.4: Sample meshes obtained during a mesh convergence procedure. The size of the elements in the meshes, from right to left, is: 0.5, 0.25, 0.12. At the interface between fiber and matrix each mesh is refined and element size is halved.

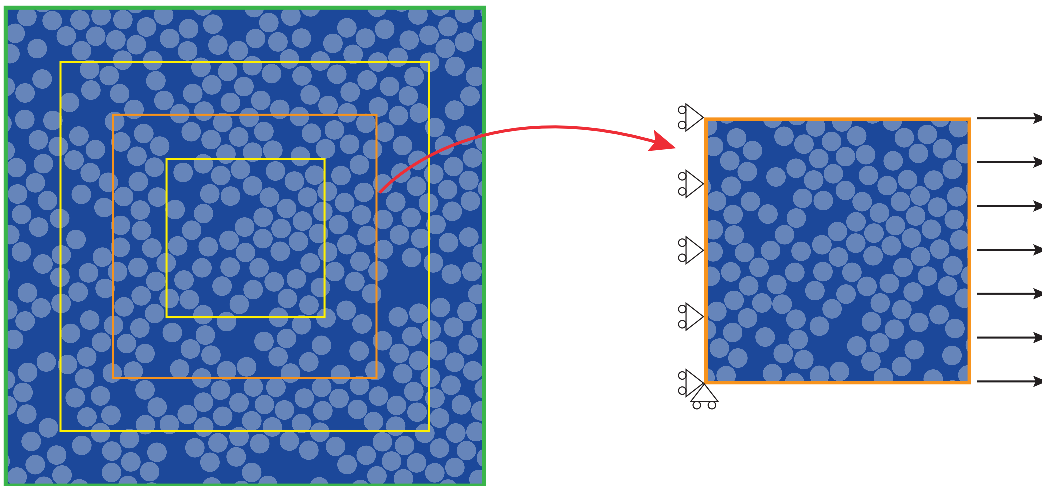

Figure 2.5: Main step (green) with other steps obtained from it. One of the yellow steps, highlighted in orange, is selected and extracted to undergo the meshing process and FEM analysis.

our RVE analysis with experimental data. For these reasons, the alternative algorithm suggested by Ge et al. [10] has been implemented and used to generate domain with V_f up to 60%. The routine used for this phase of the analysis gives as output the list of coordinates of all centers needed to reach the desired volume fraction. These data are saved in a text file that is read during the creation of the step domain meshes. In this way the positions of fibers is defined once for each sample and used throughout all its steps (see Figure 2.5). The drawback of this choice is that the desired volume fraction is actually imposed only on the main step domain. Then, V_f, for all domains (steps) that are actually used for calculation,is not strictly precise. Back to the procedure, for each step the mesh refinement is completed to obtain an accurate value of modulus. As already explained, to reach FEM convergence the implemented strategy consists of reducing the size of finite elements. To verify the convergence at least one refinement step is required, that is to say, at least two simulation needs to be executed: the first on the base mesh and the second one on the first refined mesh. Mesh element sizes are

constants for the different steps and samples. The code needs a quantitative condition to check if convergence has been reached, for this purpose the following quantity is used:

$$\Delta E_{2,i}(\%) = \frac{|E_{2,i} - E_{2,i-1}|}{E_{2,i-1}}, \tag{2.28}$$

where the index

$$i = 2, \dots, M$$

indicates the current mesh, and M is the number of available element sizes. The condition used inside the program is the following:

$$\Delta E_{2,i}(\%) < 0.0025, \tag{2.29}$$

i.e., if the relative change of E_2 from the current mesh i and the preceding mesh $i-1$ is lower than 0.25% we consider that result as converged.
The RVE analysis script obtains and saves in the output file a converged value of E_2 for each of the S steps and for each of the P samples. The data from a single sample constitutes by themselves a RVE convergence study because they are able to show that increasing the size of the domain the predicted E_2 gradually stabilizes around a certain value. Here we performed the same convergence study using a set of different samples because in this way it is possible to show that the value obtained from each RVE analysis is independent from the random distribution of fibers inside the material. This fact can also confirm in some way that it is possible to take a portion of material that is really representative of the features of the material indipendently from that particular sample. Since results from different samples are very close to each other but they still form a certain range of values, as the final answer given by the RVE analysis we are taking the mean value and the standard deviation of the main step (i.e. the largest step) of each sample. Coupling these two quantities provide a good description of the final result, giving information about the value of the estimate and its dispersion.

2.3 Python

Python [7] is the programming language selected for the implementation of the RVE analysis. This choice has been guided by two main factors: readibility and productivity. Python's design phylosophy is based on the capability of writing clear and comprehensible code. The simplicity of the language cuts down develpment time respect to other languages that are more suitable for numerical programming. The biggest drawback in using Python is, indeed, the fact that it is not as fast as other languages such as C, C++ and Fortran. The fact that Python is an interpreted language and not a compiled language has a strong impact on its performance. The decision to use Python as language for investigating code speed and optimization might seem a contradititction. However, as explained in preface, the purpose of this thesis is to inspect the performance gain and acceleration with respect to a sample basic implementation. The aim is *not* to establish a speed record for the present optimized implementation.

The most important packages used in this project are:

- *Numpy* [21], the fundamental package for numerical computations using its powerfull arrays and matrices;

- *Scipy* [28], a library that includes several algorithms and tools to manipulate numerical data in different domains;
- *meshio* [24], a package capable of handling input and output operations for many different mesh formats, including Gmsh *.msh* files;
- *tuna* [23], a modern, lightweight Python profile viewer inspired by SnakeViz.

Apart from these, other modules from python standard library are used for some particular operation like logging and profiling.

2.4 Gmsh

Gmsh [11] is an open source finite element mesh generator. In this work its role is to generate the mesh starting from RVE geometry. The geometry is created using the simple Python library called pygmsh [25], that "provide useful abstractions from the Gmsh scripting language" to easily write a `.geo` file. The OpenCASCADE kernel can use several different elements of various order. Here only 3-node triangular elements (T3) are employed since our modeling domain is 2D. The obtained mesh file (`.msh`) is passed as input to the *meshio* library that can read that file format and parse all data into a `Mesh` object. The latter is then used to get all essential data for the finite element analysis like nodal coordinates in space and element connectivity map.

2.5 Hardware

In this work two machines are used to run the simulations. One desktop workstation and a dedicated remote server. Let's see their hardware specifications in detail.
Workstation:

- Intel® Core™ i7-6700K CPU @ 4.00GHz (4 core, 8 thread),
- 16.0 GB RAM,
- OS: Windows 10 Pro (1903).

Server:

- Intel® Xeon® E5-1630 v3 CPU @ 3.70GHz (4 core, 8 thread),
- 64.0 GB RAM,
- OS: Windows Server 2016 Standard.

For all performance measurement the workstation is used, only for some RVE analysis with main step larger than 70 it is necessary to use the dedicated server due to low memory issues using the workstation.

Chapter 3

Acceleration

The acceleration procedures employed in this work mainly regard the portion of the code responsible for carrying out the Finite Element Analysis and not the RVE Analysis. This is due to the far superior applicability of techniques that speed up FEM simulations in different context. In other words, their implementation is more instructive, more useful to improve the knowledge of the finite element method.

3.1 Base Implementation

This section describe che first and simplest version of the code and use the results obtained from time profiling as guideline to turn the focus to the most heavy sections of the numerical procedures. The profiling tools used to investigate the performance of the numerical procedure are described by the end of this section. The code developed for this thesis is mainly included in a package and some scripts. The python package is called `feat` [9] and contains some modules related to the different components of the library. Functions and classes belonging to the package are used mainly by the script called `fem.py`. In this file all different functions that represent the variants of the same FEM analysis are implemented. These are then imported into the RVE analysis script that puts everything together. The base implementation is essentially the standard Finite Element Method procedure translated into Python code. The functions related to this version of the code live in the module: `base.py`. Inside this module the main function is called `assembly`; it consists on a practical level in the insertion of local $\mathbf{K}^e$ entries in their respective position of the global stiffness matrix. This process is com-

```python
def assembly(K, num_elem, elem, coord, mat_map, E_mat, h, elem_type):
    for e in range(num_elem):
        k = stiffness_matrix(e, elem, coord, mat_map, E_mat, h, elem_type)
        element_dof = compute_global_dof(e, elem, elem_type)
        for i in range(2 * elem[0].shape[0]):  # range(6) for T3
            I = element_dof[i]
            for j in range(2 * elem[0].shape[0]):  # range(6) for T3
                J = element_dof[j]
                K[I, J] += k[i, j]
    return K
```

Listing 1: Base version of the assembly function.

pleted by means of one main `for` loop over mesh elements and two more nested `for` loops that cover all entries of the local stiffness matrix. Obviously the local quantities are obtained by using equation 2.17 before their placement in the global matrix. The $\mathbf{K}^e$ matrix is computed using the function presented in Listing 1. For the simple case of the 3-node triangular element (CST), the matrices **B** and **E** are constant and the first one takes the following form:

$$\mathbf{B} = \frac{1}{2A} \begin{bmatrix} y_{23} & 0 & y_{31} & 0 & y_{12} & 0 \\ 0 & x_{32} & 0 & x_{13} & 0 & x_{21} \\ x_{32} & y_{23} & x_{13} & y_{31} & x_{21} & y_{12} \end{bmatrix}. \tag{3.1}$$

Also the thickness h is considered constant over the element so everything can be taken out of the integral in 2.17. Therefore a closed form for $\mathbf{K}^e$ has been obtained as

$$\mathbf{K}^e = A h \mathbf{B}^T \mathbf{E} \mathbf{B}, \tag{3.2}$$

where A is the area of the triangular element. The `stiffness_matrix` function returns the matrix that has to be added into the global matrix one entry at a time. For this purpose it is necessary to know the global degrees of freedom related to the local ones for the current element. The two loops with index `i` and `j` inside the main loop, scan all local entries and place them in the right position of the global stiffness matrix using the global degree of freedom numbers indicated as `I` and `J`. As already mentioned this operation is repeated for each element and the runtime directly depends on the mesh size in terms of elements numerosity. After the assembly, the problem system is ready for the application of boundary conditions and in the end for the solution. There are other operations performed at the end of the `base_analysis` function. These are related to the estimate of the transverse Young's modulus of the domain starting from the solution array of displacements. Basically the last steps are the calculation of the reactions and the evaluation of E_2 with the function `compute_modulus` that follows the procedure illustrated in Section 2.2.
The Python language comes with a built-in profiler called `cProfile`. It is a C extension that provide *deterministic profiling* with a reasonable overhead. This awesome tool can measure the time spent for the execution of each function called inside the code. This is a key step in understanding which parts of the program need an inprovement or a complete rewrite. cProfile and Tuna are employed to investigate the performance of the `base_analysis` function.
For this performance test the code is solving the elasticity problem for a domain with 50 fibers and 17820 nodes (35640 dofs). The complete runtime for this FE analysis is *169.105* seconds without the use of cProfile and about *190* seconds with its little overhead. All partial runtime data come from profiling so that they contain the overhead due to the profiling operation itself. However, all different implementations undergo the same performance test that allow us to compare them. For every performance test, only the runtime related to the call of the main fem function for that particular implementation is considered. For the base implementation this function is `base_analysis`, every other runtime measured by profiling (usually libraries initialization and import) has been neglected. The majority of runtime is spent during the execution of `numpy` linear algebra `solve` function (over 90%). The main reason for this is that all matrics are memorized in full format so the solver deals with a very large number of entries. The second call ordering by decreasing runtime is the `assembly` procedure. This step takes 2.437 seconds to complete. The speed of this basic version

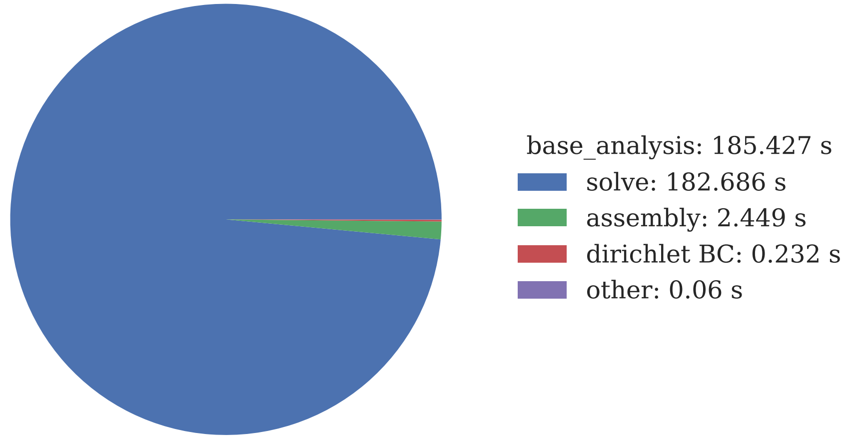

Figure 3.1: Runtime related to the main phases of the analysis represented as fractions of the total time required by the base implementation.

of the code is not satisfying because the domain used in this test is relatively small respect to the expected necessary dimension to reach RVE convergence. The RVE analysis procedure implies a series of finite element simulations to get the result. If the time for a single analysis is more than 3 minutes how long will it take to complete the whole process? It is easy to understand that a software with this performance is not so useful. Another critical issue is that for medium-sizes problems with, say, 100 fibers in the domain, which corresponds to about 7×10^4 degrees of freedom, the memory capacity (RAM) is not enough to allocate such large arrays. The default datatype for `numpy` arrays is the `float64` which needs 8 bytes for each numerical value. Using a simple expression we can get a rough estimate of the total memory required by the global stiffness matrix in full format.

$$m = n_{dof}^2 * 8\,[\text{bytes}] * \frac{1\,[\text{GB}]}{1024^3\,[\text{bytes}]} \simeq 37\,[\text{GB}] \tag{3.3}$$

The machine where all tests are executed has only 16 GB of memory so when the program try to inizialize the global stiffness matrix array the intepreter raises an error (i.e. `MemoryError` exception). The employment of matrices stored in full format has proven to be very expensive for both the resolution of the system and the memory usage.

3.2 Sparse Format for Matrices

The first step in the optimization of performance is the use of sparse format instead of full format for matrices. There are some different formats for the representation and storage of sparse matrices. The common feature between them is that they allow a substantial required memory reduction by storing only the non-zero entries. In the case of local constitutive models the stiffness matrices usually have a banded structure so they are actually very sparse, that is to say they present many zero entries. This idea is nothing innovative, indeeed it is considered a "best practice". For these reasons a sparse version of the code has been created. This variant has some differences in Dirichlet boundary conditions application, other than that regarding the replacement of full format arrays with sparse arrays. In the following listing the function `sp_assembly` is displayed and later a detailed description of the procedure is given. As suggested

```
def sp_assembly(K, num_elem, num_nodes, elem, coord, mat_map, E_mat, h, elem_type):
    data_tmp = []
    row_data = []
    col_data = []
    for e in range(num_elem):
        k = stiffness_matrix(e, elem, coord, mat_map, E_mat, h, elem_type)
        k_data = np.ravel(k)  # flattened 6x6 local matrix
        data_tmp.append(k_data)

        element_dof = compute_global_dof(e, elem, elem_type)
        row_ind = np.repeat(element_dof, 6)
        col_ind = np.tile(element_dof, 6)
        row_data.append(row_ind)
        col_data.append(col_ind)

    row = np.concatenate(row_data)
    col = np.concatenate(col_data)
    data = np.concatenate(data_tmp)
    K = sparse.coo_matrix((data,(row, col)), shape=(2*num_nodes, 2*num_nodes))
    K = K.tocsc()
    return K
```

Listing 2: Sparse version of the assembly function.

by the `scipy` documentation the best sparse format to efficiently construct finite element matrices is the Coordinate format (COO) also known as "triplet" format. In this format one entry is identified by a list including three items: row index, column index and entry value. The COO format is suitable for building the matrix but does not support arithmetic operations. Fortunately COO sparse matrices are quickly converted to other formats (CSC and CSR) that allow arithmetic operations. The procedure reproduced by the function in listing 2 is presented by Piedade Neto, Ferreira, and Proença [22] in their paper about parallelization of GFEM through multiprocessing in python. Their code has been used as trace to create an analogous function that fits well into the rest of the code. The sparse procedure maintains the same loop over elements present in the base version. This time the 6-by-6 local stiffness matrix is flattened into a one-dimensional array of 36 entries. Using global degrees of freedom row and column indices arrays related to that 36 entries. All these arrays are stored into some temporary lists (`data_tmp, row_tmp, col_tmp`). At the end of the for loop these temporary lists contain one array for each element in the mesh and they are concatenated to form a single one. These three final array are then used to construct the global sparse stiffness matrix in a single operation. The main advantage of this version over the basic one is the memory required by the global matrix K. In the base version the stiffness matrix is a square matrix stored in full format. It is possible to express the number of entries stored in memory for both the base and the sparse version of the assembly function. The expressions,found in table 3.1, are valid taking the assumption that all entries in local stiffness matrix are non-zero so that are all saved in sparse format. So for the full format matrix the number of entries depends on the square of the number of nodes while for the sparse matrix the dependance is related to the number of elements but the situation is a little bit more complex due to the presence of three arrays with different data types for the COO storage. Let's make the assumption that the number of nodes in a mesh of 3-node triangles is the double of the number of elements (i.e.

	array type	bytes	numpy data type
base **K**	2D	$d^2 n^2 * 8 = 32n^2$	float64
sparse **K**	1D (`data`)	$d^2 p^2 e * 8 = 288e$	float64
	1D (`row`)	$d^2 p^2 e * 4 = 144e$	int32
	1D (`col`)	$d^2 p^2 e * 4 = 144e$	int32
		total= $576e$	

Table 3.1: where the symbols represent the following quantities: d, the number of DOF per node (2 for 2D elasticity); n, the number of nodes in mesh; p, the number of nodes in each element (3 for CST element); e, the number of elements in mesh.

$e = 2n$). This is not exactly true but in the meshes used in this work the ratio between nodes and elements is close to two. With these assumption the relations in Table 3.1 can be expressed in terms of n and the ratio between the two memory requirements can be evaluated as

$$\frac{\text{sparse}}{\text{base}} = \frac{576\,(2n)}{32n^2} = \frac{1152n}{32n^2} = \frac{36}{n}. \tag{3.4}$$

From this last expression it is easy to see that if $n = 36$ the ratio in (3.4) is equal to 1 so that the memory required by the full format K is the same for the sparse format global matrix. This implies that starting from 36 nodes, the more n grows the more the sparse format is saving memory respect to what happens using full format. This is even clearer if we plot occupied memory as a function of the number of nodes as in Figure 3.2. In the graph the intersection between the parabola (full) and the straight line (sparse) is exactly $n = 36$. From that point on, the sparse format needs less memory. This means that the sparse format is almost always better than full format except for extremely small, and useless, problems.
The sparse implementation of the FEM analysis undergoes the same performance test used for the basic version. The domain is exactly the one employed before: 50 fibers embedded in matrix material with a total of 17820 nodes. The problem is solved in 2.230 seconds by the sparse code and in 2.552 seconds if we use cProfile to run the code. In both cases the difference with the base implementation is impressive. During the profiling of the base version it has been discovered that the most of the time has been spent for the resolution of the assembled system, here the situation is reversed.
It is evident from the graph that the majority of runtime is used for assembly procedure and only a very small fraction of the overall time is related to the execution of the sparse solver.

Even though the sparse program is much faster than the basic one in particular contexts its speed can be limiting. Even if it is not necessary a large growth in the size of the domain the problem could anyway reach very large magnitude. In our case the problem geometry and characteristics are simplified but in other situations the domain can present stress concentrations zones for its geometrical or microstructural features. In this situation could be essential to do a mesh refinement in the abovementioned zones. This operation generate an increment in the number of elements that can easily reach the order of 10^5. In this situation the time required to complete a single analysis using sparse code is around 20 seconds. For this reason the search for further performance improvements is not yet complete. A complete comparison between all the different procedures explained in this chapter is presented in Chapter 4.

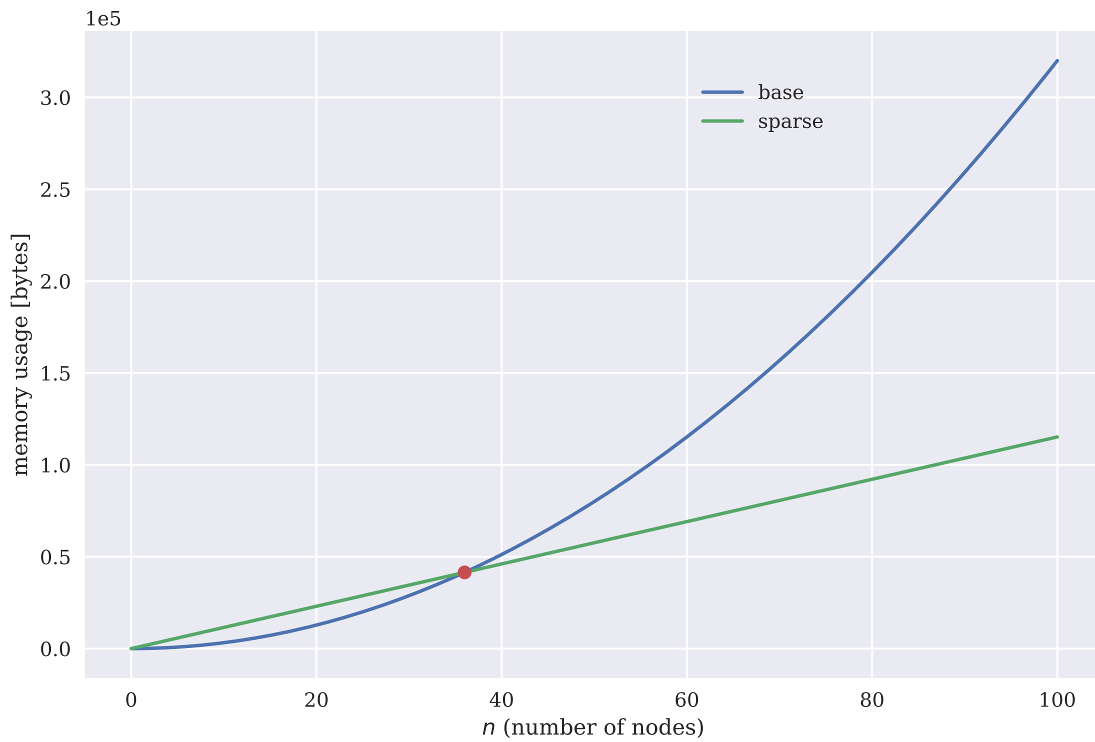

Figure 3.2: Memory usage related to **K** for base and sparse implementations.

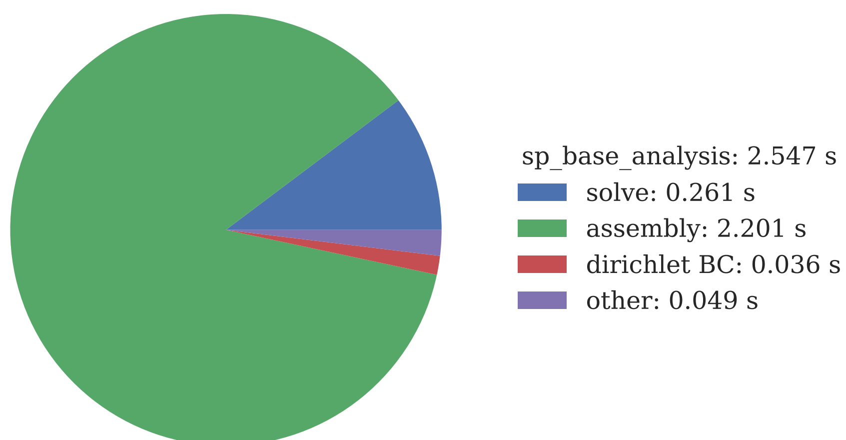

Figure 3.3: Runtime related to the main phases of the analysis represented as fractions of the total time required by the sparse implementation.

3.3 Vectorized Assembly

As seen during the profiling of the sparse function the most expensive operation performed during the analysis is the assembly. Luckily the problem of assembly efficiency has already been studied in the past taking into account also specifically the context of Python and other *"vector language"*. In particular the work of Cuvelier, Japhet, and Scarella [4] is employed in this section. In their article they define the term *"vector"* related to a programming language meaning that the language contain element-wise operations and functions on multidimensional arrays. Python does not have this feature built-in its standard library but vectorization is one of the most apppreciated and useful tools of `numpy` arrays. Vectorization is the removal of some loops from a program in favor of element-wise operations. Usually in Python and other high level programming languages the usage of loops can negatively affect the performance.

The assembly operation is usually represented by a loop over the elements that becomes very large as the mesh size is increased. It is clear that the vectorization is particularly appropriate for the purpose of improving the speed of the assembly of the stiffness matrix. In the present code the vectorized approach is contained in the `vector.py` module. The implementation is an adaptation of the algorithm OPTVS proposed in [4] and not its exact reproduction. In particular for the calculation of the entries of the stiffness matrix a different approach is followed. The description of the vectorized procedure of assembly is carried out for the 3-node triangle case. The reason for this choice lies in its simplicity ease and the direct link between code and explanation. The local **K** for the CST is a 6-by-6 symmetric matrix. During the usual assembly the local **K** is computed sequentially for every element and then assembled into the global one using global degree of freedom numbers. The key concept from the work of Cuvelier, Japhet, and Scarella [4] is that it is possible to compute a single entry of the local **K** for every element with a single vectorized operation. Let's say it is needed to compute $k_{ij} = k_{16}$ of the local matrix, once the analytical expression for each entry is available, the same matematical operations can be performed element-wise along arrays containing elements' data (i.e. connectivity table, nodal coordinates). For the 3-node triangle element it is relatively easy to obtain a closed form of the local stiffness matrix. Local entries depend only on material elasticity properties and nodal coordinates (see Equation (3.2)). The starting point for reaching a generic expression of k_{ij} comes from an article written by Griffiths, Huang, and Schiermeyer [14] in 2008. In their paper they give an integral expression for the generic entry of the local stiffness matrix. Since in Python and `numpy` arrays the numbering of item in a sequence (list, array, tuple) is zero-based, it is necessary to manipulate their expression in order to be compatible with python code indexing but the form reported here is equivalent.

$$k_{ij} = 2A\int_0^1\int_0^{1-\zeta_2} h\left(E_0\left(B_{0i}B_{0j} + B_{1i}B_{1j}\right) + E_1\left(B_{0i}B_{1j} + B_{1i}B_{0j}\right) + E_5 B_{2i}B_{2j}\right)\,\mathrm{d}\zeta_1\mathrm{d}\zeta_2 \tag{3.5}$$

The authors used this equation along with *Maple*, a CAS program, in order to solve the integral analitically. For the simple case of the CST element, as already said, that integral is trivial because B matrix and elasticity tensor are constants. So all the terms in equation 3.5 can be taken out of the integral. The remaining integral represents simply the area of the *natural* (isoparametric) element:

$$\int_0^1\int_0^{1-\zeta_2} h\,\mathrm{d}\zeta_1\mathrm{d}\zeta_2 = \frac{1}{2}h. \tag{3.6}$$

Equation (3.5) then becomes:

$$k_{ij} = hA\left(E_0\left(B_{0i}B_{0j} + B_{1i}B_{1j}\right) + E_1\left(B_{0i}B_{1j} + B_{1i}B_{0j}\right) + E_5 B_{2i}B_{2j}\right). \tag{3.7}$$

Currently, k_{ij} is expressed using the usual form for the **B** matrix (see Equation (3.1)). This form for 3-node triangle is composed of 18 entries (3 rows, 6 column), but the unique entries are only 6 of the overall. For this reason in the present implementation a different storage for these data is used. The six needed entries are saved into a smaller array **b** and the common factor $1/2A$ to all entries is treated separately.

$$\mathbf{b} = \begin{bmatrix} y_{12} & y_{20} & y_{01} \\ x_{21} & x_{02} & x_{10} \end{bmatrix} \tag{3.8}$$

Array **b** contains the same data of **B**, except for the $2A$ factor, but in less "space". Due to the shape change from **B** to **b** the creation of new indices is needed to access the correct data. Moreover the first and the second term inside parentheses in equation 3.7 are alternately zero because of the "checkered" structure of **B**. This alternating between non-zero and zero values inside **B** creates this interchange between the terms containing respectively E_0 and E_1. It is easy to check that:

$$\begin{aligned} B_{0i} &= 0 \qquad \text{if } i \text{ is odd} \\ B_{1i} &= 0 \qquad \text{if } i \text{ is even}. \end{aligned} \tag{3.9}$$

For this reason for each combination of (i, j) some B terms survive and others are null, therefore depending on the parity of (i, j) the surviving term in equation (3.7) is either the one containing E_0 or E_1. The resulting pattern is a "checkered" structure of the entries of the local stiffness matrix: E_0 and E_1 terms alternate themself as in boxes of a chessboard. To avoid the useless evaluation of terms that are known to be zero, a little bit of work is done with indices so that they can handle this "parity-dependant" behaviour. The resulting expression of the generic entry of the local stiffness matrix (k_{ij}) is the following:

$$k_{ij} = \frac{h}{4A}\left(E_n b_{AB} b_{CD} + E_5 b_{EF} b_{GH}\right) \tag{3.10}$$

where:

$$\begin{aligned} & n = \left(i + j\right) \% 2 \\ & A = i \% 2 && E = \mathtt{int}(i \% 2 == 0) \\ & B = i + (-i//2) && F = (i + (-1)^i) + (-(i + (-1)^i)//2) \\ & C = j \% 2 && G = \mathtt{int}\left(j \% 2 == 0\right) \\ & D = j + \left(-j//2\right) && H = (j + (-1)^j) + (-(j + (-1)^j)//2) \end{aligned} \tag{3.11}$$

In the preceding relation the symbols % and // indicate respectively the modulus (remainder of the division between the two operands) and the integer division (floor division). The expression $int(a == b)$ indicates a combination of two operations: first check of equality between a and b returning a boolean value (`True` or `False` in Python) and then conversion of the boolean value to integer value (`True` = 1 and `False` = 0).

At this point, Equation 3.10 can be used inside a vectorized function that permits to compute the local stiffness entry of row i and column j for all elements in the mesh with a single operation. Of course each of the local entries corresponds to a particular global entry which is characterized by a couple of global indices. Once the global indices are computed using mesh node numbering, the entries for all elements related

```
def assembly(num_elem, num_nod, elem, coord, E_array, h):
    c = coord
    e = elem
    J = X(c,e,1,0) * Y(c,e,2,0) - X(c,e,2,0) * Y(c,e,1,0)
    b = np.array([
        [Y(c,e,1,2), Y(c,e,2,0), Y(c,e,0,1)],
        [X(c,e,2,1), X(c,e,0,2), X(c,e,1,0)],
    ])
    K = sparse.csc_matrix((2 * num_nod, 2 * num_nod))

    # compute entries in the upper triangular matrix (without diagonal)
    for (row, col) in zip(*np.triu_indices(6, k=1)):
        k_data = compute_K_entry(row, col, coord, elem, b, J, E_array, h)
        row_ind, col_ind = compute_global_dof(num_elem, elem, row, col)
        K += sparse.csc_matrix(
            (k_data, (row_ind, col_ind)),
            shape=(2*num_nod, 2*num_nod),
        )

    # copy previously computed entries in the lower triangular part
    K = K + K.transpose()

    # compute the diagonal entries
    for (row, col) in zip(*np.diag_indices(6)):
        k_data = compute_K_entry(row, col, coord, elem, b, J, E_array, h)
        row_ind, col_ind = compute_global_dof(num_elem, elem, row, col)
        K += sparse.csc_matrix(
            (k_data, (row_ind, col_ind)),
            shape=(2*num_nod, 2*num_nod),
        )

    return K
```

Listing 3: Vectorized version of the assembly function.

to the local indices (i, j) can be assembled into the global sparse stiffness matrix. The following listing shows the key functions that are involved in the vectorized assembly procedure. The function `assembly` consists of 3 main steps. The first step is the calculation and assembly of all entries that belongs to the upper triangular part of the local stiffness matrix. This is done with a `for` loop in which (row, col) represent (i, j). The loop computes the data for all elements for a local entry and assembles them into the sparse matrix right away. Taking advantage of the simmetry to get the lower triangular part of the global K, it is sufficient to *"in-place"* add its transposition (2nd step). The last step is the calculation of data related to the entries inside the main diagonal. This is done following the same procedure for the upper triangular part of the matrix except that this time the loop runs over a different set of indices. At this point one might wander why for loops are still in use also in this vectorized procedure if they were depicted as something to be avoided? Here the point is that these loops iterate over a sequence of indices $(row, col) = (i, j)$ that does not depend on the number of elements in the domain but only on the type of element. In the present case of 3-node triangle the local stiffness matrix is a 6-by-6 matrix so the number of iteration is bounded by the "shape" of the local K characteristic of the actual finite element. The code inside the first loop runs 15 times to compute the data for the 15 entries in the upper triangular (excluding

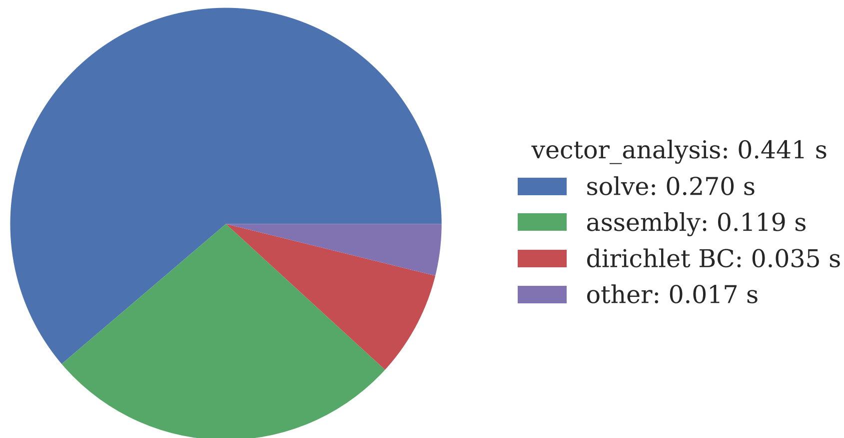

Figure 3.4: Runtime related to the main phases of the analysis represented as fractions of the total time required by the vectorized implementation.

diagonal) and the code in the second loop runs only 6 times for the 6 diagonal entries. Therefore we call the vectorized functions 21 times in total, and this number remains the same if the domain size increases or decreases. This is the reason why these loops are not a problem while the main loop employed in the base implementation is a real obstacle. To give proof of the effective advantage obtained using vectorization also this version of the FE analysis is tested solving the 50 fibers domain used also for the other implementations. The problem, characterized by 35640 degrees of freedom, is solved in 0.441 seconds. The assembly procedure and the solution of the system are the most demanding in term of runtime. In any case it is undeniable that this last version represents an additional improvement respect to the sparse implementation. Further comparisons are developed in the next chapter to sum up all differences and advantages of each variant from the previous ones.

Chapter 4

Results and discussion

4.1 FEM Acceleration

A sequence of methodologies has been added to the base version of the program. For each new feature a remarkable performance improvement has been achieved. This first section is devoted to the presentation of the numerical results in order to quantify the improvements. Often, the best way to notice and understand the meaning of numerical data is to take advantage of a graphical representation. Figure 4.1 shows the time spent to complete a single FEM analysis as function of the number of degrees of freedom in the problem. Each analysis, as explained before, gives as output the computed modulus for that particular domain. The plotted times do not include any mesh refinement. To give more realistic values of runtime also the time to load mesh data is included in these data, a time that can be non-negligible if the mesh becomes large. This is the reason why in this graph and in the following table, the reader could notice some mismatch in execution times respect to previously given data (see Chapter 3). The first thing to note is how fast the runtime of the base code increases with respect to the other versions. This is probably due to the full format of the global matrix that requires a very long time to be solved. As expected, the vectorized code is the best in terms of runtime. Also the sparse implementation performs well but the disadvantage in using it instead of the vector version is remarkable. It seems that, for sparse and vector codes, the dependency of runtime from the the number of DOFs is similar. The small difference is that vector runtime grows slightly faster than sparse runtime. The vectorized implementation runtime, for the smallest measured domain, is two orders of magnitude smaller that the basic not optimized code. To show the difference in performance between implementations even better, we present a table and a figure that represent the speedup. To measure the speedup of implementation j respect to the runtime of implementation i, the following expression is used:

$$S_{ij} = \frac{t_i}{t_j}. \tag{4.1}$$

The speedup factor of sparse and vector respect to the base implementation are very high. As described earlier the reason for this lies into the system resolution step. Between the sparse and the vector codes the global system resolution function employed is the same so the runtime of this operation should be the same in the two cases. The main improvement of the vectorized version respect to the sparse one is the optimized assembly procedure. To verifiy the actual improvement of this particular step inside

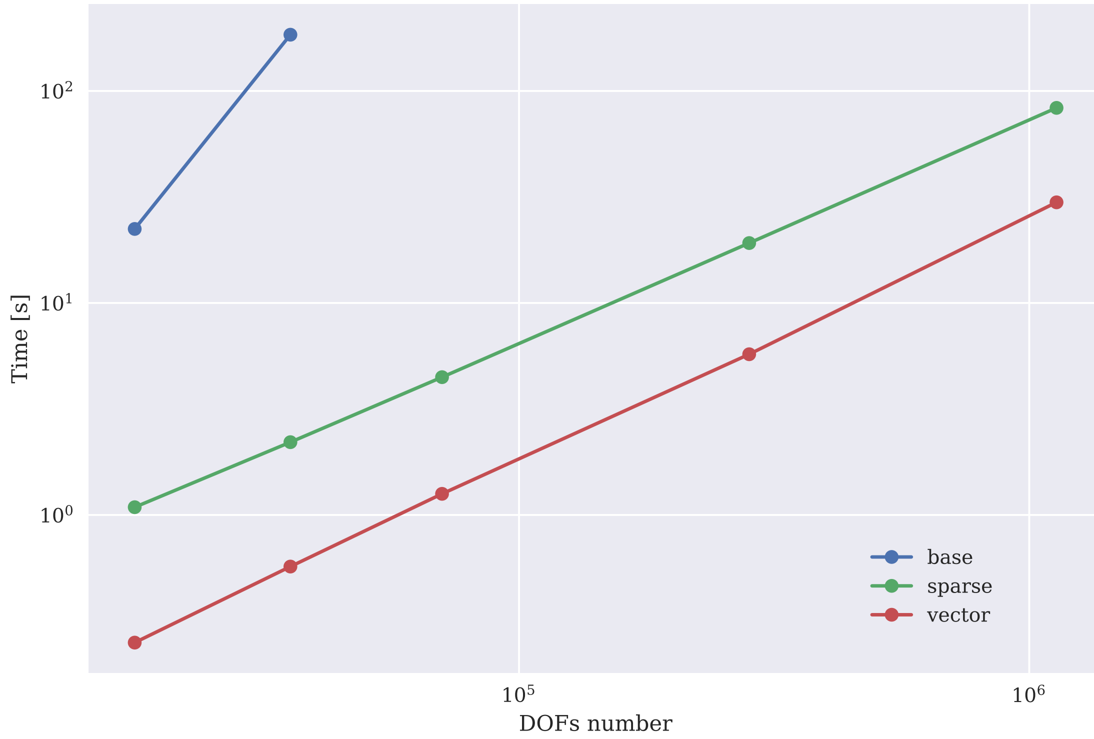

Figure 4.1: Runtime comparison between the three implementions performed for differently sized problems.

DOFs number	base (1)	sparse (2)		vector (3)		
	time [s]	time [s]	S_{21}	time [s]	S_{31}	S_{32}
17652	22.34	1.09	20.5 ×	0.25	89.36 ×	4.36 ×
35640	184.24	2.21	83.37 ×	0.57	323.23 ×	3.88 ×
70654	-	4.47	-	1.26	-	3.55 ×
282564	-	19.17	-	5.73	-	3.35 ×
1131098	-	83.22	-	29.83	-	2.79 ×

Table 4.1: Total runtime and speedup evaluated for problems of various size.

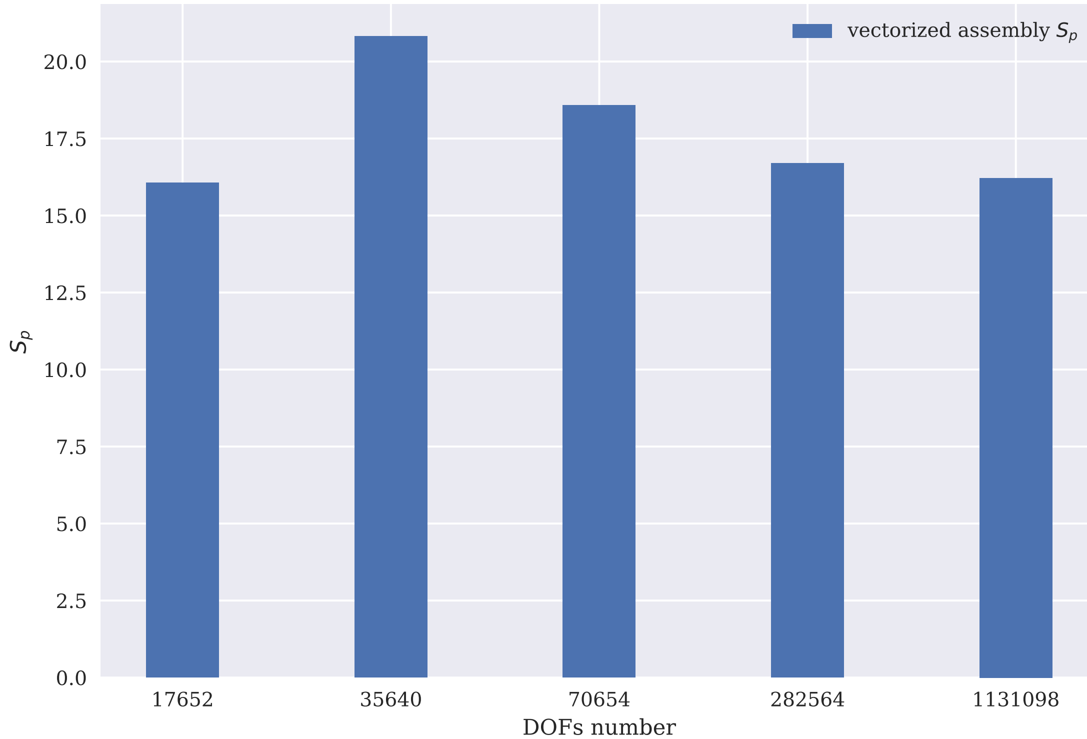

Figure 4.2: Speedup of the vectorized assembly compared with the assembly of the sparse analysis.

the complete simulation it is correct to analyze the time spent for the assembly operation. Here, the speedup of the vectorized assembly respect to the sparse assembly it is used. From Figure 4.2, the real impact of vectorization on performance is evident. For all different-sized meshes the speedup factor is almost $\times 15$. The assembly optimization is effectively resposible for the overall performance improvement showed in Figure 4.1 and in Table 4.1. The variability of the speedup factor has not been investigated but could be caused by the combination of two opposite effects. Maybe for the second mesh there is some step in the process that start to slow down the sparse version much more that the vector version. Perhaps, for larger problems this effect is gradually decreasing due to the fact that the vectorized code runtime increase a little faster than sparse code.

To establish the key differences between implementations and their feature we employ another type of comparison. This time we represent the runtime for each key phase of the FEM analysis expressed as percentage of the total runtime. The result is presented in Figure 4.3. The base implementation is completely unbalanced toward the resolution phase which occupy the most of the total runtime. As already discussed ,this is due to the extreme numerical cost related to the solution of a system in dense format. For the sparse version, the most expensive phase of the analysis is the assembly. This effect is caused by the sequential nature of the assembly that follows the standard procedure implementing the main loop over all elements. The vectorized code is the most balanced in terms of the weight that the main phases have on the overall execution time. The solver phase is the slowest one but the difference between the other steps of the analysis is smaller in this case. For all implementations the boundary condition application (Dirichlet BC in figure) require a small fraction of total runtime.

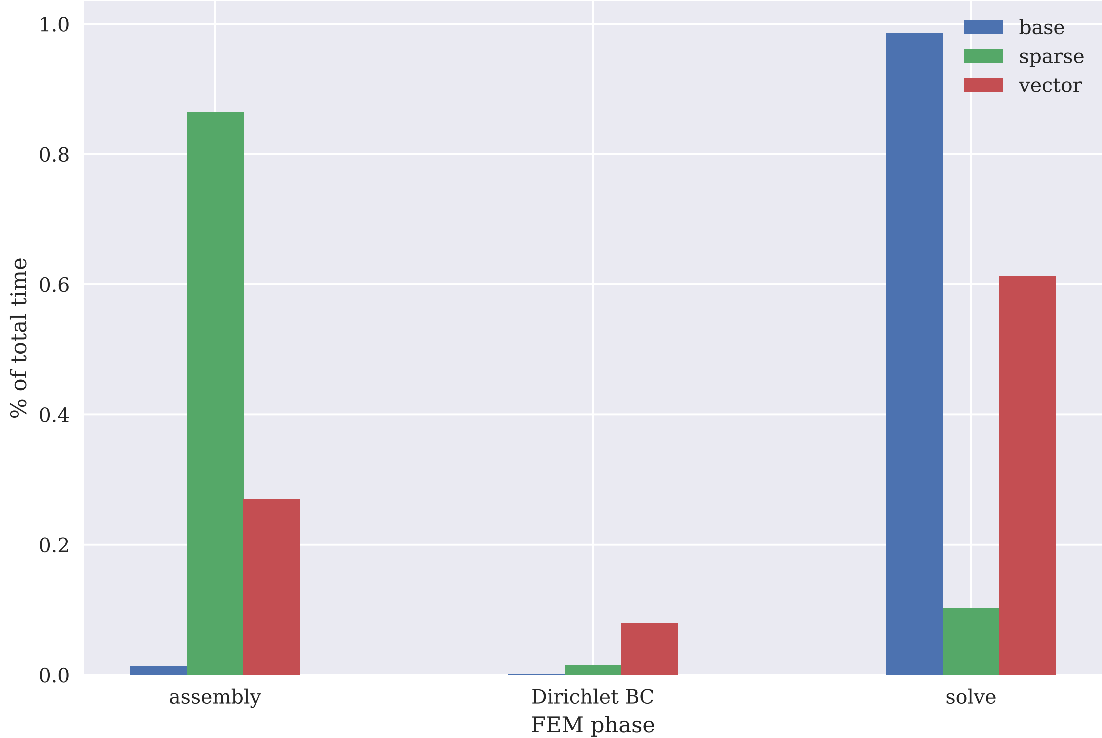

Figure 4.3: Comparison between FEM phases runtime for different implementations.

material	E [GPa]	ν
T300 carbon fiber	15.0	0.07
BSL914C epoxy matrix	4.0	0.3

Table 4.2: Mechanical properties of materials used in the RVE analysis.

4.2 RVE Analysis Results

The output of an RVE analysis, as described in Section 2.2, is the mean of all values obtained as estimated modulus for the largest domain (main step) of each sample. This mean value is provided together with the standard deviation σ of the set of values. This statistical quantity is used to quantify the dispersion of the set of E_2 estimates that are computed by the program. In the proposed RVE analyses, we used the same materials involved in the numerical verification performed by Ge et al. [10]. The materials are T300 carbon fibers and BSL914C epoxy resin as matrix. Their mechanical properties are described in Table 4.2. The RVE analysis is executed with a main step with a side lenght equal to 70 and with a fiber volume fraction Vf of 30%. In this case the RVE analysis is set up to compute 5 samples, each consisting of 5 steps (domains). The output data set is plotted in Figure 4.4. Every sample present a convergence trend moving from small RVE domain to large RVE side. The values to which each RVE convergence sample tends are very close to each other. This is also confirmed by the quantities displayed inside the box: the standard deviation of the data is very limited. The low dispersion of final step data in some way confirms the fact that the size of the final RVE is enough for it to be representative. The mechanical behavior resulting from the analysis demonstrates to be independent from the arrangement of fibers, indeed we

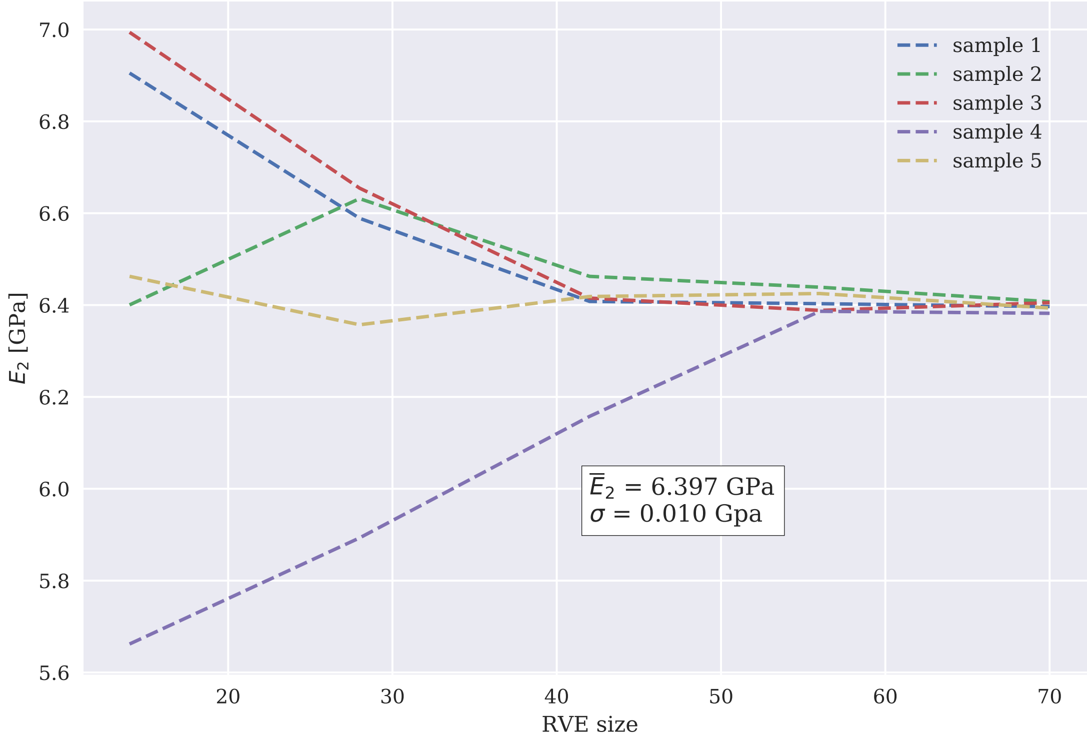

Figure 4.4: RVE analysis for composite material with $V_f = 30\%$ and a 70×70 main step.

observe the same tendency for all samples. Ge et al. [10] in their paper display also experimental data for the same combination of fiber and matrix but with $V_f = 60\%$. To compare the our result with experimental values we execute the analysis with the same materials , volume fraction. In this case the main step side lenght is 100. In Figure 4.5 we display the 5 steps that belong to the first sample. All domains are represented with the same size for clarity but their true dimensions are progressively increased from 20 (label 1) to 100 units (label 5). The resulting data from this last run of the program are displayed in Figure 4.6. The last step data for all samples are very close to each other. This is not enough to affirm that the analysis reached a good convergence. The downward trend of the last part of the curves suggests that the convergence value has not yet been reached. A possible explanation for this behavior is that the volume fraction affects the size of the domain. Seemingly, the current maximum RVE size (100x100) is not enough for the domain to be representative. Unfortunately, the available machine does not allow us to explore larger RVE size due to memory limitations. Then, we anyway compare the result with experimental result considering that convergence has not been reached for it. Moreover, the comparison is extended also to result obtained by Ge et al. [10] and to analytical models described in 1.2. The numerical verification in [10] is completed using a 3D FEM model along with volume average approach. E_2 values and the percentage variations Δ with respect to the preceding rows are summarize in Table 4.3. It is evident that there is a certain difference (-19.2%) between the result of the RVE analysis and the experimental value. The reasons for this inaccurate result are probably related to the simplifications applied to the problem during the analysis. Here, it is important underline that also the result obtained with the average volume procedure followed by Ge et al. [10] has a certain error with respect to the experimental measure. The same occurs for the Reuss model and the Halpin-Tsai model: the former

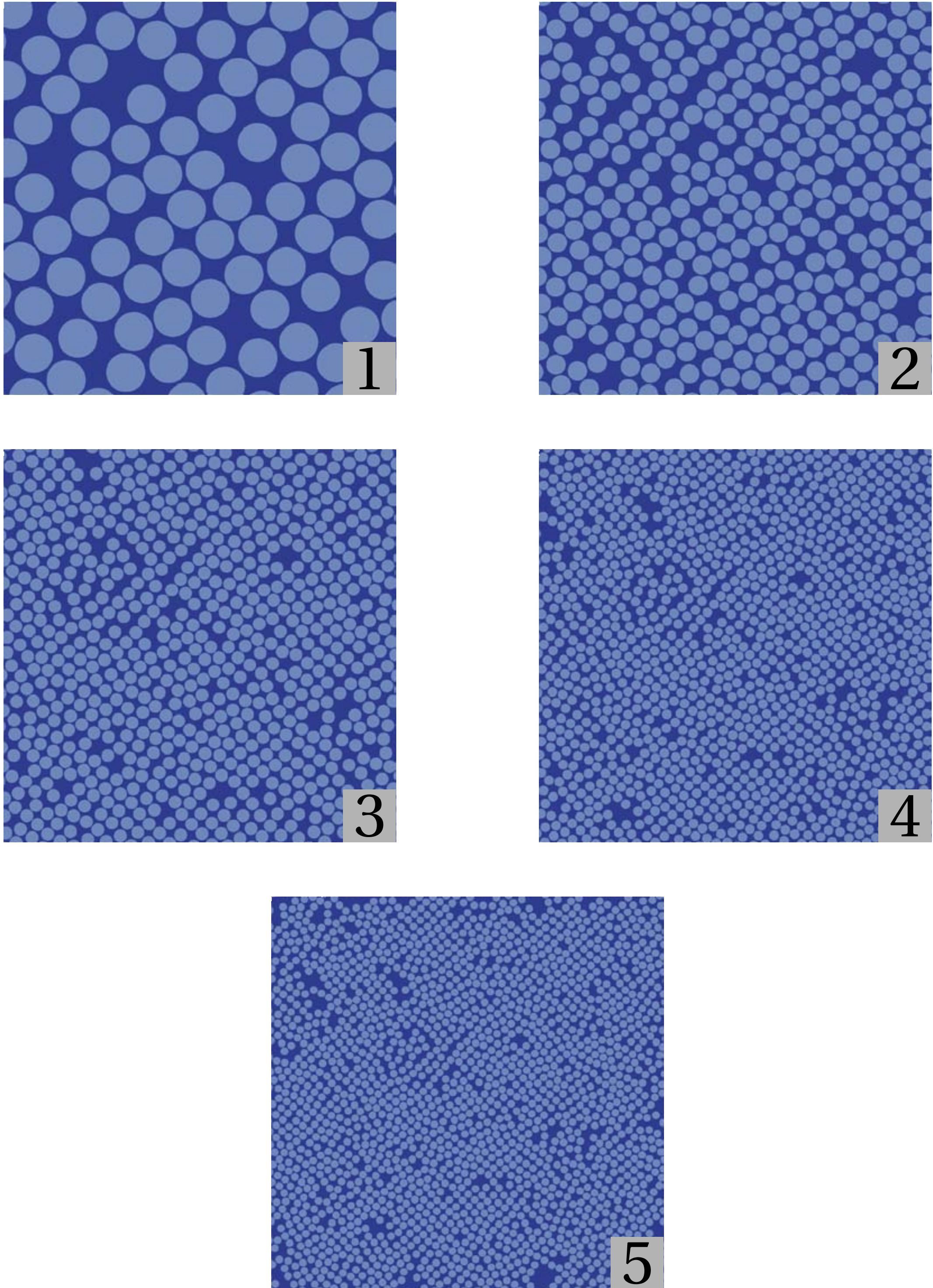

Figure 4.5: Steps generated for the first sample of the RVE analysis. The domain sizes are 1) 20, 2) 40, 3) 60, 4) 80, and 5) 100.

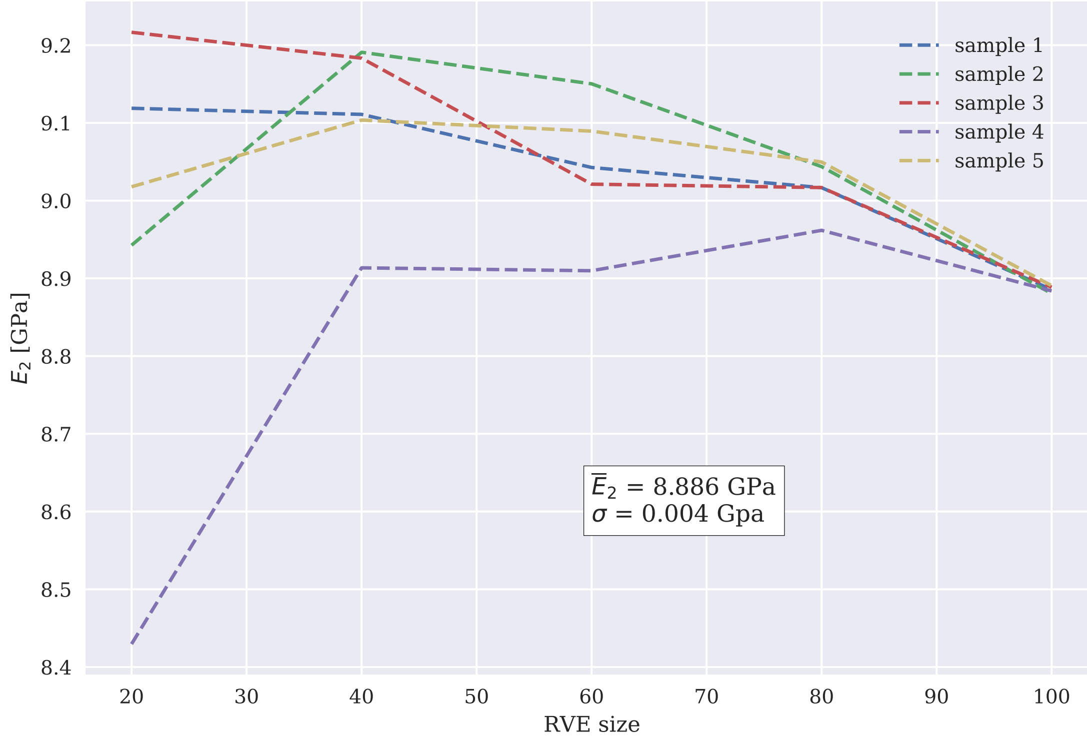

Figure 4.6: RVE analysis for composite material with $V_f = 60\%$ and a 100×100 main step

has a strong difference from the experimental value, for the latter the discrepancy is similar to that of the RVE analysis. A more detailed discussion of the causes leading to this result is reported in the next section.

4.3 Conclusions

The actual RVE procedure used in this project has been choosen mainly for its implementation simplicity. Other methods found in literature like homogenization and volume averaging allow to obtain the complete elasticity matrix and they are much more complex than our technique. The simplicity of our method has a cost in terms of accuracy. The RVE procedure presented in this work provide only a rough estimate of

	E_2 [GPa]	Δ% (a)	Δ% (b)
Experimental	11.0	-	-
Reuss model	7.143	-35.1%	-
Halpin-Tsai model	8.829	-19.7%	-
Ge-Wang FEM	9.838	-10.6%	-
`feat` FEM RVE	8.886	-19.2%	-10.7%

Table 4.3: Summary of transverse modulus experimental data and estimates obtained from different procedures. Moreover, Percentage variations with respect to experimental data (a) and with respect to Ge et al. [10] (b) are computed.

the transverse modulus. This aspect it has been clear since the beginning. The result obtained from the RVE analysis should be taken as an indicative estimate of the value that would have been obtained using more appropriate numerical methods. The idea of selecting domains with different size starting from the same portion of material used by Terada et al. [26], seemed very close to a realistic situation. The difference of the RVE result with respect to the experimental data is probably related to a number of different factors. First, the present model is two-dimensional while the real sample used in experimental test is obviously three-dimensional. However, the dimensionality can't be the only cause of the mismatch and this is proven by the fact that also the three-dimensional model of Ge et al. [10] is characterized by a certain error. Another important effect that is not taken into account by our model is the presence of defects. Inside a real specimen there are a lot of different defects. The interface between fiber and matrix is usually not perfect. This means that it presents debonded zones where the link between fiber and matrix is weak or completely broken. Another phenomenon is the formation of small voids near the interface of inside the matrix. Often fibers are not perfectly aligned. All these effects create a situation in which the materials that form the composite are far from being homogeneous. However, in our model the properties of the matrix and the fiber are considered uniform. This important difference and the fact that our modeling does not take into account the factor listed above, makes the observed mismatch perfectlThis meay understandable. To address this kind of inaccuracy a complete study of the statistical distributions of defects and irregularities would have been necessary. For each parameter, e.g. fiber arrangement, interface quality, defect distribution, a separate study would have helped to identify which of them has a relevant effect on the modulus. This kind of study is however not among the objectives of this work. As stated in the preface one of the main goals of this project is the acceleration of a basic FEM code.

After the discussion of performance data made in Section 4.1, we can safely say that the results are positive. The sparse version and the vectorized version bring an obvious improvement in performance compared to the starting code. The vectorization of the assembly from Cuvelier, Japhet, and Scarella [4] demonstrates to be at least 15 times faster than the standard sequential assembly (see Figure 4.2). The vectorized code can analyze domains with more than 1 million degrees of freedom in about 30 seconds (see Table 4.1), which is an extremely satisfying result. As already explained in Section 2.3, the Python language is not the best choice if performance is critical. However, Python is used in different fields for prototyping the software at first and later translate it in another faster language. The main reason behind this practice is the ease that characterize Python. In other word, this language allows the user to focus on what to do, instead of how to do it. From this perspective the `feat` [9] package developed within this project can be seen as a prototype code for future development. But before the translation of `feat` in another language there are other acceleration techniques that can be explored. Just to give a couple of quick examples: Python multiprocessing capabilites and Just In Time (JIT) compilation provided by `Numba` package [20].
Recalling the preface, the main purpose of this thesis is to learn something new. This task has been certainly accomplished during the months of work on this project. From technical knowledge of Finite Element Method to workflow organization, I think everything I've had to deal with could come in handy in the future.

Bibliography

[1] D. F. Adams and D. R. Doner. “Longitudinal shear loading of a unidirectional composite”. In: *Journal of Composite Materials* 1.1 (1967), pp. 4–17.

[2] D. F. Adams and D. R. Doner. “Transverse normal loading of a unidirectional composite”. In: *Journal of composite Materials* 1.2 (1967), pp. 152–164.

[3] R. D. Cook et al. *Concepts and Applications of Finite Element Analysis.* John Wiley & Sons, Inc., 2001.

[4] F. Cuvelier, C. Japhet, and G. Scarella. “An efficient way to assemble finite element matrices in vector languages”. In: *BIT Numerical Mathematics* 56.3 (2016), pp. 833–864.

[5] C. Dong. “Effects of Process-Induced Voids on the Properties of Fibre Reinforced Composites”. In: *Journal of Materials Science & Technology* 32.7 (2016), pp. 597–604. ISSN: 1005-0302. DOI: `https://doi.org/10.1016/j.jmst.2016.04.011`. URL: `http://www.sciencedirect.com/science/article/pii/S100503021630038X`.

[6] W.J. Drugan and J.R. Willis. “A micromechanics-based nonlocal constitutive equation and estimates of representative volume element size for elastic composites”. In: *Journal of the Mechanics and Physics of Solids* 44.4 (1996), pp. 497–524. ISSN: 0022-5096. DOI: `https://doi.org/10.1016/0022-5096(96)00007-5`.

[7] Python Software Foundation. *The Python Programming Language.* URL: `https://www.python.org/`.

[8] R. L. Foye. “An evaluation of various engineering estimates of the transverse properties ofunidirectional composites”. In: *Proceedings of the Tenth National SAMP E Symposiwn-Advanced Fibrous Reinforced composites* (1966), pp. 9–11.

[9] P. Fumiani. *basic-ph/feat.* URL: `https://github.com/basic-ph/feat`.

[10] W. Ge et al. “An efficient method to generate random distribution of fibers in continuous fiber reinforced composites”. In: *Polymer Composites* 40.12 (2019), pp. 4763–4770. DOI: `10.1002/pc.25344`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/pc.25344`.

[11] C. Geuzaine and J. Remacle. “Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities”. In: *International Journal for Numerical Methods in Engineering* 79.11 (2009), pp. 1309–1331. DOI: `10.1002/nme.2579`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2579`.

[12] R. F. Gibson. *Principles of Composite Material Mechanics.* 4th Edition. Boca Raton: CRC Press, 2016. DOI: `https://doi.org/10.1201/b19626`.

[13] M. Goudarzi and A. Simone. “Fiber neutrality in fiber-reinforced composites: Evidence from a computational study”. In: *International Journal of Solids and Structures* 156-157 (2019), pp. 14–28. DOI: https://doi.org/10.1016/j.ijsolstr.2018.07.023.

[14] D. V. Griffiths, J. Huang, and R. P. Schiermeyer. “Elastic stiffness of straight-sided triangular finite elements by analytical and numerical integration”. In: *Communications in Numerical Methods in Engineering* 25.3 (2009), pp. 247–262. DOI: 10.1002/cnm.1124. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.1124.

[15] J. C. Halpin. *Effects of environmental factors on composite materials*. Tech. rep. AFML-TR-67-423. Air Force Materials Lab Wright-Patterson AFB OH, June 1969.

[16] L. R. Herrmann and K. S. Pister. “Composite properties of filament-resin systems”. In: *ASME Paper* 63-WA (1963), p. 239.

[17] R. Hill. “Elastic properties of reinforced solids: Some theoretical principles”. In: *Journal of the Mechanics and Physics of Solids* 11.5 (1963), pp. 357–372. ISSN: 0022-5096. DOI: https://doi.org/10.1016/0022-5096(63)90036-X. URL: http://www.sciencedirect.com/science/article/pii/002250966390036X.

[18] R. M. Jones. *Mechanics Of Composite Materials*. 2nd Edition. Boca Raton: CRC Press, 1999. DOI: https://doi.org/10.1201/9781498711067.

[19] S. Kari, H. Berger, and U. Gabbert. “Numerical evaluation of effective material properties of randomly distributed short cylindrical fibre composites”. In: *Computational Materials Science* 39.1 (2007), pp. 198–204. DOI: https://doi.org/10.1016/j.commatsci.2006.02.024.

[20] S. K. Lam, A. Pitrou, and S. Seibert. “Numba: A LLVM-Based Python JIT Compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

[21] T. E. Oliphant. *A guide to NumPy*. USA: Trelgol Publishing, 2006.

[22] D. Piedade Neto, M. D. C. Ferreira, and S. P. B. Proença. “Generalized finite element method computation: parallelization using python multiprocessing package”. In: *Mecánica Computacional* (2011).

[23] N. Schlömer. *nschloe/tuna*. URL: https://github.com/nschloe/tuna.

[24] N. Schlömer et al. *nschloe/meshio v4.0.8*. Version v4.0.8. Feb. 2020. DOI: 10.5281/zenodo.3691940. URL: https://doi.org/10.5281/zenodo.3691940.

[25] N. Schlömer et al. *nschloe/pygmsh v6.1.1*. Version v6.1.1. Apr. 2020. DOI: 10.5281/zenodo.3764683. URL: https://doi.org/10.5281/zenodo.3764683.

[26] K. Terada et al. “Simulation of the multi-scale convergence in computational homogenization approaches”. In: *International Journal of Solids and Structures* 37.16 (2000), pp. 2285–2311. DOI: https://doi.org/10.1016/S0020-7683(98)00341-2.

[27] S. Torquato, O. U. Uche, and F. H. Stillinger. “Random sequential addition of hard spheres in high Euclidean dimensions”. In: *Phys. Rev. E* 74 (6 Dec. 2006), p. 061308. DOI: 10.1103/PhysRevE.74.061308. URL: https://link.aps.org/doi/10.1103/PhysRevE.74.061308.

[28] P. Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: https://doi.org/10.1038/s41592-019-0686-2.

[29] G. Zhang and S. Torquato. “Precise algorithm to generate random sequential addition of hard hyperspheres at saturation”. In: *Physical Review E* 88.5 (2013), p. 053312.