Rui Amendoeira Esteves

Master of Science

Python-based MEMS inertial sensors design, simulation and optimization

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Micro and Nanotechnologies Engineering

Adviser: Michael Kraft, Full Professor, KU Leuven

Co-adviser: Joana Vaz Pinto, Invited Assistant Professor, NOVA University of Lisbon

Examination Committee

Chair: Prof. Dr. Hugo Manuel Brito Águas
Rapporteur: Prof. Dr. Manuel João de Moura Dias Mendes
Member: Prof. Dr. Michael Kraft

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2020

Python-based MEMS design, simulation and optimization

This document was created using the (pdf)LaTeX processor, based on the NOVAthesis template, developed at the Dep. Informática of FCT-NOVA by João M. Lourenço.

Scientia potentia est.

Acknowledgements

Obtaining a master degree is, so far, the greatest achievement in my life. So, I could not let this moment pass without acknowledging the people and institutions that supported me during these years and, in fact, made this possible.

The first acknowledgement must go to Professor Rodrigo Martins and Professor Elvira Fortunato for creating a truly pioneer degree in Portuguese science, effectively establishing Portugal in the nanotechnology area, and allowing students to be a part of science's future.

I would like to express my gratitude towards my adviser Professor Michael Kraft for taking me aboard the MNS team, allowing me to do research on a topic of significant interest, and providing a very positive work experience and environment. I would also like to thank Professor Joana Pinto for always being available and for the guidance, which was of great help in the writing of this work.

To Chen Wang, for all the hours, great guidance and friendship during my time in Leuven. To Mathieu and Sina, for all the precious advice and fun at the office. To the rest of the MNS team, for the warm reception, and for the excellent work environment you created.

Furthermore, I would like to thank all my colleagues that shared these five years with me. A special acknowledgement to Bernardo Madeira, Diogo Carvalho and José Barnabé - our time in Segundo Esquerdo was one of the best things I take from these years. Sharing this experience with you was a pleasure, and I will always be grateful for it.

To Eduardo Oliveira, who didn't live in the house, but was an essential part of it and a great friend. To Gui, Dmytro, Mariana Tomé, Raquel, Mariana Abreu, Maria Francisca, Guida, Alentejano, Diogo Lopes, Eduardo Encarnação and André Alves, for all the memories and help. To Pipa and Miguel, for all the advice, dinners, and very important guidance.

I want to thank my friends Madeira, Ticas, Zé Diogo, Diogo, Esha, Chico, Cacelas, Honório, Sousa, Ospital, Crua, Cunha, Tiago, and Manel. We have been together since elementary school, and the influence you have had in my life has been so significant and overwhelmingly positive that it is impossible to express by words. Thank you for all the support and countless memories, I could not be more proud and grateful to call you friends.

I would like to express my enormous gratitude to my family. To my mother and father, for all the love, for always being present, for supporting me in every way, for always believing in me, and for always putting us first - you will always be an example to me. Thank you for all your sacrifices and efforts to provide for me. To my sister, for always being ready to defend your beliefs and to think for yourself, thus forcing me to be in touch with different points of view. Aos meus avós, pelo vosso amor e tempo. Pela inexplicável dedicação e devoção para com os filhos e netos, por tudo o que me ensinaram e deram ate hoje - o meu muito obrigado, que nunca será suficiente. All of you (and Rocky) contributed to create the best family environment I could ever wish for, thank you.

Lastly, I want to express my deepest gratitude towards my girlfriend Maria. Having you in my life has been the greatest gift I have ever got, and I am nothing but thankful for having you as a colleague, friend and girlfriend. Thank you for always keeping me in the right track, for always believing in me, and for facing this Belgian adventure with me.

Abstract

With the rapid growth in microsensor technology, a never-ending range of possible applications emerged. The developments in fabrication techniques gave room to the creation of numerous new products that significantly improve human life. However, the evolution in the design, simulation, and optimization process of these devices did not observe a similar rapid growth. Thus, the microsensor technology would benefit from significant improvements in this domain.

This work presents a novel methodology for electro-mechanical co optimization of microelectromechanical systems (MEMS) inertial sensors. The developed software tool comprises geometry design, finite element method (FEM) analysis, damping calculation, electronic domain simulation, and a genetic algorithm (GA) optimization process. It allows for a facilitated system-level MEMS design flow, in which electrical and mechanical domains communicate with each other to achieve an optimized system performance. To demonstrate the efficacy of the co-optimization methodology, an open-loop capacitive MEMS accelerometer and an open-loop Coriolis vibratory MEMS gyroscope were simulated and optimized - these devices saw a sensitivity improvement of 193.77% and 420.9%, respectively, in comparison to its original state.

Keywords: Microelectromechanical systems (MEMS), inertial sensors, Python, finite element method, genetic algorithm, optimization, accelerometer, gyroscope

Resumo

Com o rápido crescimento observado na tecnologia de micro sensores, emergiu um vasto numero de aplicações possíveis. Os desenvolvimentos que ocorreram nas técnicas de micro e nano fabricação deram lugar à criação de um grande número de novos produtos que melhoram significativamente a vida humana. No entanto, a evolução no processo de design, simulação e optimização destes dispositivos não acompanhou o progresso previamente mencionado. A tecnologia dos micro sensores beneficiaria, então, de um desenvolvimento significativo neste domínio.

Este estudo apresenta uma nova metodologia para a co-optimização electromecânica de microssistemas electromecânicos (MEMS) para sensores de inércia. O software desenvolvido é composto por um bloco para design de geometria, outro bloco para análise com método de elementos finitos (FEM), um *script* de cálculo de amortecimento, uma simulação da interface electrónico e um processo de optimização pelo algoritmo genético (GA). Esta ferramenta permite um processo de design de MEMS facilitado, no qual os domínios electrónico e mecânico comunicam entre si para atingir um sistema final optimizado. Para demonstrar a eficácia da metodologia de co-optimizacao, um acelerómetro capacitivo MEMS e um giroscópio vibratório de Coriolis MEMS foram simulados e optimizados - estes dispositivos viram a sua sensibilidade aumentada em 193.77% e 420.9%, respectivamente, em relação ao seu estado original.

Palavras-chave: Microssistemas electromecânicos (MEMS), sensores de inércia, Python, método dos elementos finitos, algoritmo genético, optimização, acelerómetro, giroscópio

Contents

List of Figures

List of Tables

Acronyms

CAD computer-assisted design

DoF degree of freedom

FEA finite element analysis
FEM finite element method
FOM figure of merit

GA genetic algorithm
GPS global positioning system

MEMS microelectromechanical system

PDEs partial differential equations

VDC vehicle dynamic control

Symbols

V_{AC}	alternate-current voltage
P_a	ambient pressure
A	area
V_{C2V}	voltage from the conversion of capacitance
C_{bottom}	capacitance between the bottom electrode and the proof-mass
C_{top}	capacitance between the top electrode and the proof-mass
C_{int}	reference capacitance
$c_{squeeze}$	squeeze damping coefficient
c_{slide}	slide damping coefficient
m_m	mass of the moving structure
$:$	colon operator
m_C	mass subjected to Coriolis force
ΔC_s	capacitance variation from the sensor
F_c	damping force
V_{DC}	direct-current voltage
$\{U\}$	displacement function
$disp$	displacement
u	displacement vector field
d	distance between electrodes
x_0	drive amplitude
ω_D	drive mode frequency
ω_S	sense mode frequency
$\mu_{eff_{squeeze}}$	effective viscosity for squeeze film damping
$\mu_{eff_{slide}}$	effective viscosity for slide film damping
$F_{balanced}$	electrostatic force from balanced actuation
F_{comb}	electrostatic force from comb fingers

λ	eigenvalue
ω	eigenfrequency
ϵ_0	free space permittivity
ϵ	symmetric strain-rate tensor
ϵ_r	relative permittivity of the dielectric medium
f	body force per unit of volume
I	identity tensor
K_n	Knudsen number
λ_L	Lamé parameter λ
L	length of the comb fingers
$[M]$	mass matrix
y_0	mechanical scale factor
λ_f	mean free path
Δf	frequency mismatch
μ_L	Lamé parameter μ
μ	mean viscosity of the medium
n	outward pointing unit normal at the boundary
∇	divergence operator
N	number of comb fingers
Ω	body domain
Ω_Z	angular-rate around the z-axis
a	overlapping area between electrodes and proof-mass
c	ratio between the width and length
Q_{factor}	quality factor
m_S	moving mass in the sense mode
Q_{sense}	quality factor of sense mode
σ	stress tensor

σ_s	squeeze number
$[K]$	stiffness matrix
t	thickness
tr	trace operator
v	test function
$\hat{V}$	vector-valued test function space
V_{cm}	reference voltage
V_{DD}	supply voltage
V	voltage

Chapter 1

Motivation and objectives

In the past decade, inertial sensor technology has suffered a rapid market growth: smartphones and tablets, gaming systems, virtual reality equipment, toys, and power tools are good examples of the wide adoption these devices have seen on consumer electronics products [1]. Most people already carry a microelectromechanical system (MEMS) inertial sensor in their pockets - the ordinary smartphone combines gyroscopes and accelerometers in order to provide the user with a global positioning system (GPS), a rotation detector and velocity measurements.

However, these mundane implementations are not alone in the inertial sensor world - the automotive [2], aerospace [3], and military industry make use of these devices in numerous applications. Apart from the standard GPS, modern vehicles now possess a vehicle dynamic control (VDC) system that helps the automobiles with regaining control in the event of skidding [4]. In the military industry there are ongoing efforts to integrate MEMS inertial sensors with projectiles and aircraft, providing a chance to measure in-flight dynamics [5].

In order to design, simulate, and optimize these devices, engineers have divided the process into two very distinct - yet symbiotic - domains: mechanical and electrical. This workflow is often severely divided, and usually a great deal of simplification is applied to one of the domains in order to achieve a complete simulation and optimization of the other one [6]. Moreover, the typical design methodology combines multiphysics software and a programming language interpreter program. The fact that there is a very restricted set of tools to choose from, combined with the need for compatibility between different commercial software, dramatically limits the potential for customization and adaptation to specific designs.

This work aims to develop a novel co-simulation and co-optimization process for MEMS devices fully based on Python, in order to provide a complete open-source solution that provides an insight into both the mechanical and the electrical domains, while paying attention to their interaction.

Chapter 2

Work strategy

This work followed three consecutive phases:

1. The initial step was to develop the Python program. Initially, research was conducted to find similar work as well as to search for software libraries that could be of use in this endeavour. Then, a geometry builder part, a finite element method (FEM) block, and an electrical script were built. Finally, a genetic algorithm (GA) was designed and connected to the program. The four sections assemble a complete general-purpose MEMS simulation and optimization software.

2. The second step encompassed designing, simulating, and optimizing a capacitive MEMS accelerometer with the new program. The simulations are compared with commercial multiphysics software, and the optimization result is analyzed. This step represents the program's first implementation on a MEMS inertial sensor.

3. The third step was to test the software with a MEMS gyroscope. This design represented a remarkable challenge for the program to process, simulate and optimize - thus, making for a reliable way to validate further the system when applied to inertial sensors.

Chapter 3

Introduction

Microelectromechanical Systems (MEMS) are defined as a combination of electrical and mechanical systems at micrometer scale. These devices are fabricated using photolithography methods. This technology allows for the fabrication of moving microstructures on a substrate, allowing for the creation of remarkably complex structures which turn into mechanical and electrical systems [7].

3.1 MEMS inertial sensors

MEMS inertial sensors are a group of sensors that measure acceleration or angular motion - the first is referred to as accelerometer and the second as gyroscope. With the advent of the micromachining technology [8], the production costs for these devices decreased enough to oversee their expansion into consumer applications. Previously confined to cost-heavy industries, such as military and aerospace, inertial sensors rapidly grew into many other areas such as automotive, biomedical, navigation, and smart systems [9].

One of the most prominent applications for micromachined accelerometers is in the automotive industry [10]. A study on the perspectives of MEMS sensors by Senturia *et al.* [11] stated that the silicon accelerometer dominates the market for automotive airbag deployment, a life-saving mechanism responsible for avoiding 2790 deaths per year in the United States of America alone [12]. MEMS accelerometers applications in this area include crash and skid detection - both crucial when designing stabilization systems [13].

In the biomedical industry, there is a growing interest in the integration of MEMS accelerometers into various applications. Kusmakar *et al.* [14] demonstrated the potential to build an ambulatory monitoring convulsive seizure detection system, using an accelerometer. Fall detection systems using MEMS accelerometers are being widely adopted [15, 16]. Van Thanh *et al.* [17] developed a prototype for fall detection that alerts an emergency contact in case the elderly person has an accident. Furthermore, fitness trackers comprising MEMS accelerometers are now popularly used by the general public [18].

Similarly, micromachined gyroscopes are widely adopted in the automotive industry [2] - seeing applications in rollover protection, stability and active control systems, and inertial navigation. A gyroscope detects the angular rate of a car, and if this value hits a critical threshold - a safety system will adjust the steering wheel and brakes to prevent the vehicle from overturning [9]. Another common application for MEMS gyroscopes is platform stabilization - using these sensors to detect an angular motion and automatically adjust a platform such as a video camera or robotic arm, to achieve a stable surface [19].

Inertial sensors have a bright future ahead of them with an endless array of possibilities. Thus, it is of the most significant interest to research and develop new ways of designing, simulating and optimizing MEMS inertial sensors. Novel methodologies of co-simulation can open doors to news designs and applications, as well as creating a more efficient work-flow.

3.1.1 Accelerometers

An accelerometer is a sensor which can detect acceleration. The general working principle of this device is described as a body which suffers a detectable displacement when it is under an external acceleration force. Despite the large number of accelerometers types, the vast majority has a proof-mass attached to a reference frame by a suspension system, illustrated in Figure 3.1 - this mechanical structure is designed to move along a specific axis, in order to detect acceleration in this direction [9].

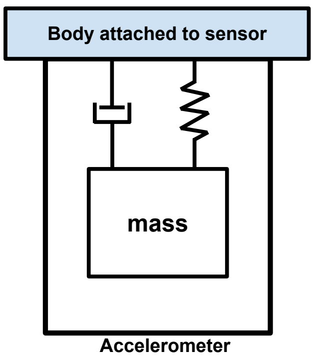


Figure 3.1: Lumped model of accelerometer attached to a body

The deflections are transformed into an electrical signal. This process of transducing can take three forms: resistive interfaces, piezoelectric interfaces, and capacitive interfaces [20]. In this work, the studied inertial sensors have a capacitive transducing interface - these devices comprise a set of one or more fixed electrodes and one or more moving electrode. The movement caused by an acceleration modifies the distance between electrodes, provoking a capacitance change which is then captured by a readout circuit.

3.1.2 Gyroscopes

A gyroscope is a sensor that measures the angular rate of an object - the rate of rotation. There are three types of gyroscopes: spinning mass, optical, and vibrating gyroscopes [21]. For micromachined gyroscopes, the most common approach to sense an angular rate is to use vibrating mechanical elements. This type of devices involve no rotating parts which endure friction and wear, allowing for a successful miniaturization under micromachining techniques. Vibrating gyroscopes induce and detect Coriolis force in order to measure the angular motion [7].

The Coriolis force is a fictitious force that emerges from the Coriolis effect, which only acts on an object when the motion is observed from a rotating non-inertial reference frame. Jean Bernard Léon Foucault demonstrated this phenom in 1851, with the Foucault pendulum [22]: when a swinging pendulum attached to a rotating platform is observed by a stationary observer from above - the pendulum oscillates along a straight line; however, an observer in the rotating platform would see that the line precesses. That precession can only be described with dynamic equations if the Coriolis force is included [7, 23].

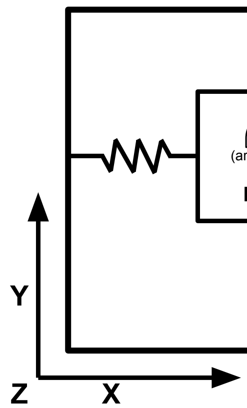


Figure 3.2: Working principle of MEMS coriolis vibratory gyroscope

Coriolis vibratory gyroscopes comprise an inertial mass element and a suspension system that keep the proof-mass suspended above the substrate. The sensitive element is driven to oscillation along one axis with known amplitude (driving mode), when the device rotates around another axis, the Coriolis effect causes the proof-mass to move in an orthogonal direction (sensing mode). In this study, the displacement in the sense mode produces a detectable capacitance change [24]. This working principle is illustrated in Figure 3.2, in which the gyroscope detects angular motion around the z-axis.

3.2 Genetic Algorithm

When Charles Darwin, in the nineteenth century, revolutionized science by discovering the processes by which nature selects and evolves its organisms [25], it was not possible to foresee the numerous applications his discoveries would inspire. In the same century, Gregor Mendel laid down the bases of genetic inheritance [26] - complementing Darwin's findings. The genetic algorithm was designed, taking the aforementioned principles as inspiration, making use of computational resources to optimize all kinds of devices and processes.

The genetic algorithm applies evolution principles to a set of individuals: the algorithm runs through several generations, starting from an initial population with initial parameters and defined fitness goals, and letting the best individuals survive and reproduce themselves - mixing the parameters of the ancestors with random mutations, imitating the natural process until the population converges to a higher performance state [27].

In this study, the genetic algorithm is applied to MEMS inertial sensors, setting electro-mechanical performance parameters as fitness goals in order to achieve a complete co-optimization.

3.3 MEMS design, simulation and optimization

The design, simulation, and optimization process of a MEMS device is illustrated in Figure 3.3, and can be broadly described by the sequence: design of initial geometry, mechanical parameter simulation and optimization, design of electrical interface, and simulation of complete system [28].

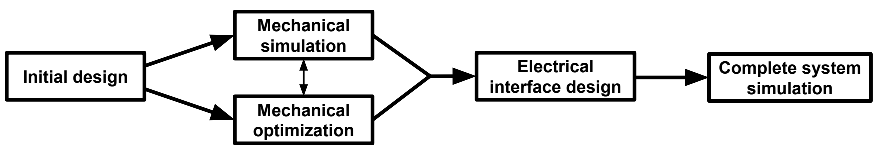


Figure 3.3: General MEMS design, simulation and optimization process-flow

In order to design, simulate and optimize MEMS inertial sensors, engineers have separated the process into two very distinct - yet symbiotic - domains: mechanical and electrical. This workflow is often severely divided, and usually, a great deal of simplification is applied to one of the domains to achieve a complete simulation and optimization of the other.

MEMS mechanical structures are usually designed in a computer-assisted design (CAD) software and commonly comprise thousands of degree of freedom (DoF) which lead to a high computational cost when simulating mechanical behaviour. To bypass this obstacle, engineers have used reduced-order modelling methods to build system-level models [29], bringing the thousands of DoF down to a few, with the three DoF being the most basic option, frequently used when designing closed-loop control systems. For example, Hung *et al.* [30] used low-order models to improve simulation time significantly, while Kudryatsev *et al.* [31] tried to achieve a reduced-order model for a MEMS piezoelectric energy harvester, and Nayfeh *et al.* [32] developed two reduced-order models for MEMS applications.

The aforementioned method can be helpful when designing and optimizing the sensor's electrical interface; however, it fails to take into consideration the full complex mechanical structure, and consequently, the interaction between electrical and mechanical domains.

Moreover, the typical optimization methodology combines multiphysics software and a programming language interpreter program. Wang *et al.* [33] presented a MEMS mechanical optimization method that allows for the generation of freeform geometries - combining COMSOL [34] finite element analysis and modelling with a GA implemented in MATLAB [35], demonstrating its effectiveness with the optimization of a MEMS accelerometer. Solouk *et al.* [36] used the same methodology to optimize a MEMS gyroscope concerning an automotive application.

Although this approach takes into consideration the complex mechanical structure of MEMS as well as its electrical interface, it does possess several limitations. It does not fully capture the interaction between mechanical and electrical domains, and the fact that there is a very restricted set of tools to choose from, combined with the need for compatibility between different commercial software ends up limiting the potential for customization and adaptation to specific designs.

Chapter 4

Simulation Methodology

4.1 Finite Element Method

In this work, the finite element method (FEM) is implemented in the developed software, as it is a fundamental mathematical tool to simulate mechanical structures. It is chosen over the finite difference method due to its ability to handle complex geometries. It is also used by COMSOL [34], a simulation multiphysics software employed to provide a viable comparison and validation means.

The finite element method approaches any problem by subdividing a continuous entity into finite smaller parts, solve each one individually, and reassemble them as seen on Figure 4.1. It is a mathematical tool necessary to apply finite element analysis (FEA) to a physical phenom. Using numerical methods to deeply comprehend any phenomena is essential, mostly because many of the physical processes, and indeed the vast majority of solid mechanics ones, are described by partial differential equations (PDEs) [37].

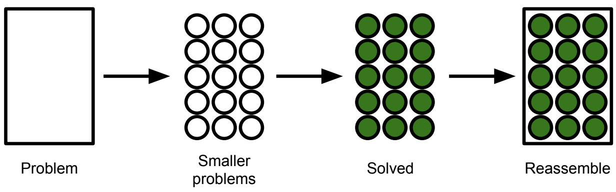


Figure 4.1: The finite element method approach

For a computer to solve PDEs, it applies the FEM to divide an extensive system into smaller subparts - finite elements. This division is called space discretization, and the generation of a mesh achieves it - this is a way of transcribing a 2D or 3D object into a series of mathematical points that can be analyzed.

There are several categories of PDEs: elliptic, hyperbolic, and parabolic [38]. This study focuses on solid mechanics simulation, thus, the main category applied to this area is elliptic, which can be solved using a variational method - FEM.

A variational method has its basis on the principle of energy minimization: when a boundary condition (e.g. displacement) is applied, the configuration where the total energy is minimum is the one that prevails. The process of solving these equations with this method starts with multiplying the PDEs by a test function, then integrate the resulting equation over the domain, and finally perform integration by parts with second-order derivatives. The unknown function to be approximated is named trial function. The trial and test functions belong to a particular function space, which specifies the functions properties as well as the spatial domain in which they act [39].

4.2 Python language and libraries

The Python [40] programming language was used to develop the co-simulation and co-optimization system in this study. Python is a high-level language exceptionally well suited for scientific and engineering environments - its highly modular nature and clean syntax provide a simple and direct code writing suitable in many scientific applications [41].

The language's open source license allows the user to use, sell, and distribute any developed Python-based application with no need for special permissions. Its ability to interact with a wide range of other software, to run on a significant number of platforms, to allow realtime code development without the need to compile every time it changes, makes the language a powerful candidate for scientific computing and application development [42].

The advantages mentioned above contribute heavily to the selection of Python for this work. However, the main reason for this choice lies in the countless number of library modules provided either officially from Python or from the global developer's community, which opens the door to a never-ending number of possible combinations and applications. A software was developed in Python to simulate and optimize MEMS devices. It used different modules and libraries like *pygmsh*, *meshio*, and *FEniCS* to generate and simulate 3D geometries.

The *pygmsh* [43] module was used to build the desired geometries. It combines *Gmsh* [44], a finite element mesh generator and Python to create a versatile tool which can create complex geometries with code. The *meshio* [45] module was implemented on the software in order to perform mesh import and export operations. It is useful to allow information processing between the various program parts and provides the possibility to visualize generated meshes.

The *FEniCS* [46] library is a powerful open-source computing platform for PDEs. It enables users to translate scientific models into efficient finite element code [47] and supports parallel processing - which allows for a significant computation speed increase. The software used this module to perform FEM simulation.

Chapter 5

Results and discussion

5.1 Python simulation and optimization software

A complete MEMS co-simulation and co-optimization program comprises different essential blocks. As introduced in Chapter 2, this type of software needs to englobe:

1. A geometry designer or processor with meshing abilities.
2. A FEM simulation block powerful enough to process different mesh sizes with varying degrees of complexity.
3. A personalized electrical domain script capable of interpreting the mechanical results for each MEMS device.
4. A GA section that takes into consideration both mechanical and electrical performance parameters.

A software covering all the aforementioned abilities was developed with a general structure depicted in Figure 5.1. This program also englobed a viscous damping calculation, which the genetic algorithm takes into consideration. In this way, the optimization considers an important parameter that is a direct result of the interaction between mechanical and electrical domains.

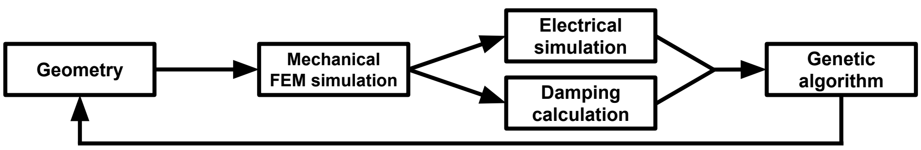


Figure 5.1: General block diagram for the developed software.

5.1.1 MEMS geometry design in Python

A competent geometry design and meshing system requires several fundamental characteristics: the ability to create different shapes and perform boolean operations on them (union, difference, intersection, and complement), the capacity to create a customizable mesh to be assigned to the created geometry, and the possibility to import and export files.

For this purpose, as previously mentioned in Chapter 4.2, two Python libraries were chosen - *pygmsh* and *meshio*. *Pygmsh* is used for geometry building and mesh generation, next the *meshio* library is applied to generate a file that can be read by the other software blocks. This implementation is shown in Listing I.1 and II.1.

5.1.2 FEM simulation for displacement and modal analysis

The FEM block is the most complex and vital simulation in the program, arguably the core of the software. In this study, as stated in Chapter 4.1, this tool is used to solve PDEs problems involving linear elasticity equations: the first is to calculate the displacement resulting from an applied force, and the second is to perform a modal analysis - obtaining the MEMS eigenfrequency modes.

5.1.2.1 Displacement analysis

For MEMS inertial sensors simulation, PDEs modelling small deformations of elastic bodies become the main mechanical objects of study [20]. When a force is applied to a body Ω, the equations describing the suffered deformations on isotropic elastic conditions are the following [48]:

$$-\nabla \cdot \sigma = f \text{ in } \Omega \tag{5.1}$$

$$\sigma = \lambda_L \operatorname{tr}(\epsilon)\, I + 2\mu_L \epsilon \tag{5.2}$$

$$\epsilon = \tfrac{1}{2}(\nabla u + (\nabla u)^T) \tag{5.3}$$

In these equations: ∇ represents the divergence operator [49], σ is the stress tensor [50], f stands for the body force per unit of volume, λ_L and μ_L denote the Lamé's elasticity parameters regarding the body's material [51], I signify the identity tensor, *tr* represents the trace operator (on a tensor), ϵ stands for the symmetric strain-rate tensor, and u is the displacement vector field.

Combining (5.2) and (5.3) yields

$$\sigma = \lambda_L(\nabla \cdot u)I + \mu_L(\nabla u + (\nabla u)^T) \tag{5.4}$$

As referred in Section 4.1, the variational formulation of (5.1 - 5.3) begins with the inner product of equation (5.1) and a test function $v \in \hat{V}$, where $\hat{V}$ stands for a vector-valued test function space, and integrating it over the domain Ω.

$$-\int_\Omega (\nabla \cdot \sigma) \cdot v \,\mathrm{d}x = \int_\Omega f \cdot v \,\mathrm{d}x \tag{5.5}$$

The expression $\nabla \cdot \sigma$ includes second-order derivatives that belong to the unknown u, so the term that contains it is integrated by parts.

$$-\int_{\Omega} (\nabla \cdot \sigma) \cdot v \, \mathrm{d}x = \int_{\Omega} \sigma : \nabla v \, \mathrm{d}x - \int_{\partial\Omega} (\sigma \cdot n) \cdot v \, \mathrm{d}s \tag{5.6}$$

The colon operator represents the inner product of two tensors, and n is the outward pointing unit normal at the boundary. The expression $\sigma \cdot n$ is known as the stress vector and regularly designates a boundary condition - it is assumed that it is prescribed on a part $\partial\Omega_T$ of the boundary as $\sigma \cdot n = T$, where T is a constant. On the rest of the boundary, the value of the displacement is represented as a Dirichlet [52] condition. Thus, it is obtained:

$$\int_{\Omega} \sigma : \nabla v \, \mathrm{d}x = \int_{\Omega} f \cdot v \, \mathrm{d}x + \int_{\partial\Omega_T} T \cdot v \, \mathrm{d}s \tag{5.7}$$

Replacing the stress tensor, σ from (5.4) in the previous equation (5.7) produces the variational form with u as unknown. It is now possible to summarize the variational formulation - find $u \in V$ in a manner that

$$a(u,v) = L(v) \quad \forall v \in \hat{V} \tag{5.8}$$

In which

$$a(u,v) = \int_{\Omega} \sigma(u) : \nabla v \, \mathrm{d}x \tag{5.9}$$

$$L(v) = \int_{\Omega} f \cdot v \, \mathrm{d}x + \int_{\partial\Omega_T} T \cdot v \, \mathrm{d}s \tag{5.10}$$

In the colon operator product $\sigma : \nabla v$, if ∇v is represented as a sum of its symmetric and anti-symmetric parts, the remaining part will be the symmetric one because σ is a symmetric tensor. This allows for the replacement of ∇v by the symmetric gradient $\epsilon(v)$:

$$a(u,v) = \int_{\Omega} \sigma(u) : \epsilon(v) \, \mathrm{d}x \tag{5.11}$$

The equation 5.11 is the one that clearly emerges from the principle of energy minimization applied to potential elastic energy and it is the one implemented on the developed software, as seen on Listing I.2 and II.2. In this work, the MEMS designer inputs the required boundary conditions and the program takes into consideration those constants to evaluate aforementioned equation.

The software needs to possess the inertial sensor's material properties in order to process the devices response properly. Single crystal silicon is one of the most common materials used in inertial sensors, and it was the one chosen in this study. The properties Silicon crystal (100) are listed in Table 5.1 [53].

Table 5.1: Material properties of Silicon crystal (100) [53]

Material property	Value
Young's Modulus	131 GPa
Poisson ratio	0.28
Density	2330 kg/m^3
Lamé parameter λ_L	8.452×10^{10} N/m^2
Lamé parameter μ_L	6.641×10^{10} N/m^2

5.1.2.2 Modal analysis

To study MEMS inertial sensors, the knowledge of natural frequencies and the corresponding mode shapes of a device during free vibration becomes essential. Modal analysis is performed to calculate said properties, making use of FEA. The method solves an eigensystem - a group of all eigenvectors belonging to a linear transformation matched with the corresponding eigenvalue [54], the first represents the mode shape and the second represents the frequency.

The eigenvalue problem solved in this software is the following [55]:

$$[K]\{U\} = \lambda[M]\{U\} \tag{5.12}$$

In this equation, $[K]$ stands for the stiffness matrix - obtained from the assembly of equation 5.11, $[M]$ represents the mass matrix - assembled, taking into account the device's geometry and material density, $\{U\}$ symbolizes the displacement function, and λ is the desired eigenvalue.

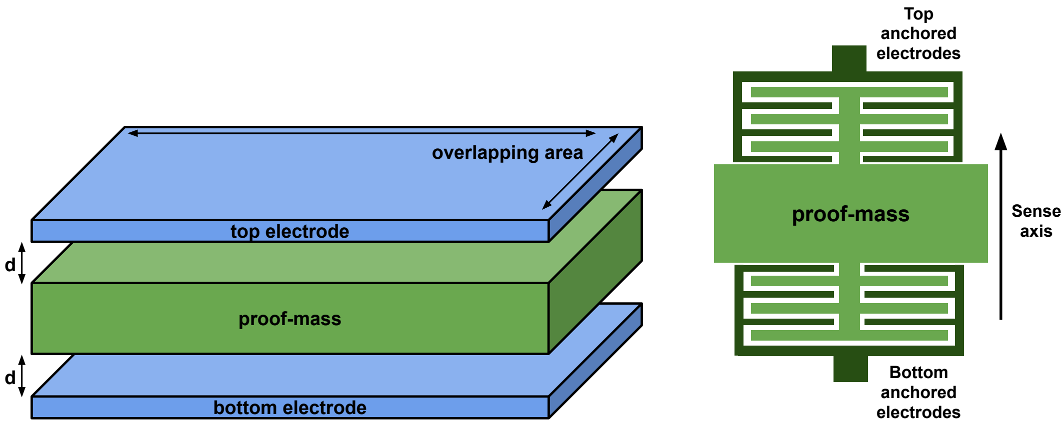
With the help of PETSc linear algebra package to assemble the matrices and of SLEPcEigenSolver to compute the solution to said matrices, the software can now return the eigenfrequencies related to the obtained eigenvalues: $\lambda = \omega^2$. The eigenvectors associated with the mentioned eigenvalues are exported to a file, which can be opened in ParaView [56] to visually analyze the mode shapes. The modal analysis implementation in this software is shown in Listings I.3 and II.3.

5.1.3 Electronic domain simulation

The electronic domain simulation block is the third and most variable block in the program. Every MEMS architecture possess a different system, thus requiring a different script for each design - this allows for total customization and an accurate simulation.

In this work, the analyzed devices were MEMS inertial capacitive sensors - as introduced in Subsection 3.1.1, these sensors produce a detectable signal when a displacement causes a capacitance change between parallel plates. This general principle leads to different capacitance sensing mechanisms which have to be taken into account when implementing the software.

Two MEMS inertial sensors were simulated: a capacitive accelerometer and a linear vibratory gyroscope, as shown in Figure 5.2.



(a) MEMS accelerometer principle (case study 1) (b) MEMS gyroscope principle (case study 2)

Figure 5.2: Capacitive sensing structures present in both case studies

The simulated MEMS capacitive accelerometer (Figure 5.2a) has its proof-mass between two electrodes, which results in differential sensing defined by equations (5.13),(5.14) [57]. In these Equations, ϵ_0 stands for free space permittivity and ϵ_r represents the relative permittivity of the dielectric medium - these are listed in Table III.1. The initial vertical distance between the proof-mass and the electrodes is designated as d, and their overlapping area is a.

$$C_{top} = \frac{\epsilon_0 \epsilon_r a}{d - disp}, C_{bottom} = \frac{\epsilon_0 \epsilon_r a}{d + disp} \tag{5.13}$$

$$\Delta C = C_{top} - C_{bottom} \tag{5.14}$$

$$\tag{5.15}$$

The linear vibratory gyroscope comprises a differential sensing mechanism as well, seen in Figure 5.2b: the proof-mass contains two sets of comb-fingers which move along the y-axis, altering the gap between the fixed electrodes and the moving ones - the capacitance change produced by this system is described by Equations (5.16),(5.17). In the aforementioned equations, N stands for the number of comb fingers, t represents the thickness, L signifies the comb fingers length, and d is the gap between said fingers.

$$C_{top} = N\frac{\epsilon_0 tL}{d-disp}, C_{bottom} = N\frac{\epsilon_0 tL}{d+disp} \tag{5.16}$$

$$\Delta C = C_{top} - C_{bottom} \approx 2N\frac{\epsilon_0 tL}{d^2}disp \tag{5.17}$$

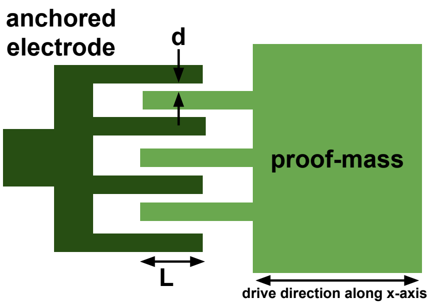
In order to drive the gyroscope's proof-mass into driving resonance, electrostatic actuation is applied. Electrostatic actuators rely on the force between two electrodes when a voltage is applied between them [58]. Parallel-plate actuation electrodes are commonly built to apply a force in a specific direction - aligned with the desired motion direction and DoF of the target mass.

In the designed MEMS gyroscope, the implemented mechanism is the balanced actuation. The interdigitated comb-drives seen in Figure 5.3a generate the desired force by sliding parallel to each other; this force is described by Equation 5.18. A balanced actuation system illustrated in Figure 5.3b works by applying $V_1 = V_{DC} + V_{AC}$ to one set of electrodes, and $V_2 = V_{DC} - V_{AC}$ to the opposing set. This method allows a linearization of the force regarding a constant voltage V_{DC} and a varying V_{AC} - the electrostatic force is finally described by Equation 5.19 [7].

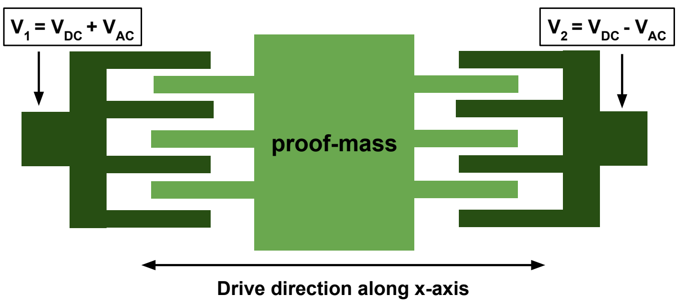
$$F_{comb} = \frac{-\epsilon_0 t}{d} N V^2 \tag{5.18}$$

$$F_{balanced} = 2\frac{\epsilon_0 LtN}{d^2} V_{DC} V_{AC} \tag{5.19}$$

The aforementioned sensing and actuating equations are inserted in the software, providing the possibility to study the effects of different potential on the drive mode, and enabling the user to pass electrical performance parameters onto the genetic algorithm - thus, achieving a co-optimization.



(a) Variable-area actuator principle



(b) Balanced actuation scheme

Figure 5.3: Electrostatic actuation system present in MEMS gyroscope (case study 2)

A capacitance to voltage converter is implemented in the code, in order to generate a readable electronic signal from the provoked capacitance change. The converter is a simplified version of the converting block designed by Utz et al. [59], seen in Figure 5.4 with a governing equation stated in equation 5.20.

$$V_{C2V} = \frac{2\Delta C_s \cdot (V_{DD} - V_{cm})}{C_{int}} \tag{5.20}$$

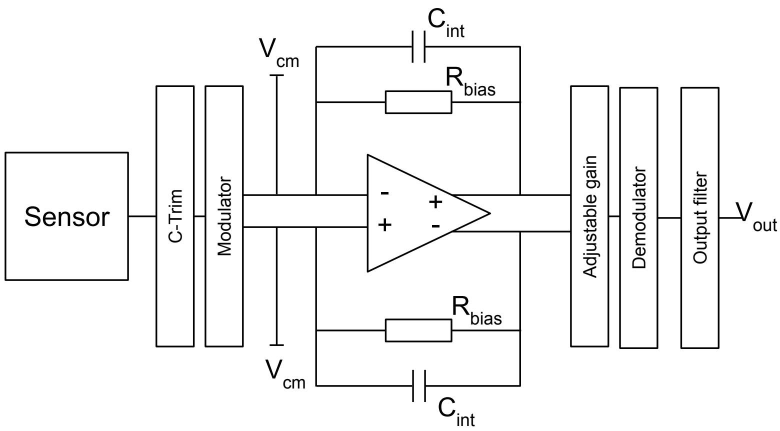


Figure 5.4: Block diagram of capacitance to voltage converter circuit implemented with both MEMS devices

In Listing I.4 and II.5, it is possible to see the implementation of the electrical domain simulation.

5.1.4 Damping calculation

Damping stands for the energy loss effects in any oscillatory system. In this study, the modelled damping was the viscous damping. This type of damping mechanism occurs when the gas surrounding the vibratory structures presents viscous effects caused by the internal friction of the gas trapped in the middle of vibratory structures such as comb fingers.

Viscous damping is the primary contributor to overall damping and poses as an essential parameter to be estimated [60]. In this study, squeeze film damping and slide film damping were modelled to calculate the drive and sense mode's quality-factor of both simulated devices. The simulated MEMS inertial sensors were considered to be surrounded by air ($\epsilon_r = 1$).

When two parallel plates move towards each other, they squeeze the fluid (in this case, air) between them - creating a squeeze film damping phenom, seen in Figure 5.5a. Additionally, slide film damping exists when two plates slide parallel to each other - illustrated in Figure 5.5b. To model these two effects, the following equations were implemented:

$$K_n = \frac{\lambda_f}{d} \tag{5.21}$$

$$\mu_{eff_{squeeze}} = \frac{\mu}{1+9.638K_n^{1.159}} \tag{5.22}$$

$$\frac{F_c}{z} = \frac{64\sigma_s P_a A}{\pi^6 d}\sum_{m,n\,odd} \frac{m^2+c^2n^2}{(mn)^2[(m^2+c^2n^2)^2+\sigma^2/\pi^4]} \tag{5.23}$$

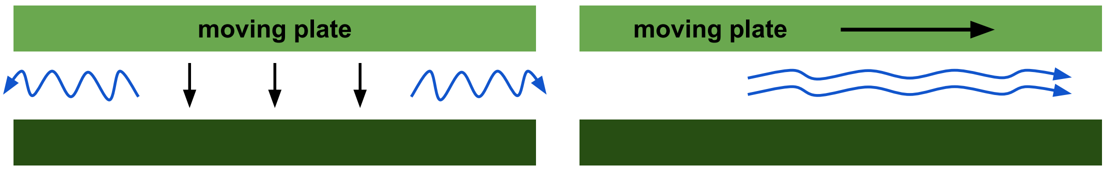
$$\sigma_s = \frac{12\mu_{eff_{squeeze}} w^2}{P_a d^2}\omega \tag{5.24}$$

$$c_{squeeze} = F_c \cdot N_{combs} \tag{5.25}$$

$$\mu_{eff_{slide}} = \frac{\mu}{1+2K_n+0.2K_n^{0.788}e^{-K_n/10}} \tag{5.26}$$

$$c_{slide} = \mu_{eff_{slide}} \frac{A}{d} \cdot N_{combs} \tag{5.27}$$

$$Q_{factor} = \frac{m_m \cdot \omega}{c} \tag{5.28}$$



(a) Squeeze film damping effect (b) Slide film damping effect

Figure 5.5: Viscous damping effects modelled in the software

In Equation 5.21, K_n represents the Knudsen number which is a measure of gas rarefaction effect - the ratio of the mean free path λ_f to the gap d containing the gas [61]. For squeeze film damping modelling, Equation 5.22 denotes the effective viscosity in which μ stands for the mean viscosity (in this study, air) [62]; Equation 5.23 represents the squeeze film damping force in which z is the plate deflection, P_a is the ambient pressure, for a plate with width w and length l - $A = wl$ and $c = w/l$, σ is the squeeze number [63]; Equation 5.24 stands for the aforementioned squeeze number in which ω denotes frequency; Equation 5.25 represents the viscous film damping coefficient for a sensing structure with a number of comb fingers N_{combs}.

For slide film damping modelling, Equation 5.26 refers to the effective viscosity associated with this type of damping [64], and Equation 5.27 describes the lateral damping coefficient in which A denotes the overlap area of the plates.

Equation 5.28 represents the quality factor [65, 66] associated with a movement mode, in which m stands for the mass of the moving structure, ω denotes the frequency of vibration mode, and c is the damping coefficient associated with the movement type.

The damping calculations are implemented in the software as seen on Listing I.5 and II.4.

5.1.5 Genetic algorithm optimization

The GA block is the responsible for the electro-mechanical co-optimization. As explained in Section 3.2, the algorithm starts with an initial set of individuals, calculates the figure of merit (FOM) of each one, selects the top performers, reproduces and mutates them, and then forms the new generation.

In this study, the genetic algorithm was developed with the following workflow, demonstrated in Figure 5.6:

1. The first step of the algorithm is to initialize a first generation with 100 individuals containing the initial geometric parameters listed in each case study's section.

2. A calculation of each device's FOM is then executed - in the first generation this attribute is equal for all individuals.

3. To assemble the next generation, both an integral copy and a mutated copy of the 25 best devices are placed in the population - the remaining 50 devices are randomly mutated.

4. This process is repeated for a designated number of generations - until half of the population converges to a high-performance FOM and an individual is selected.

The genetic algorithm implementation is displayed in Listing I.6 and II.6.

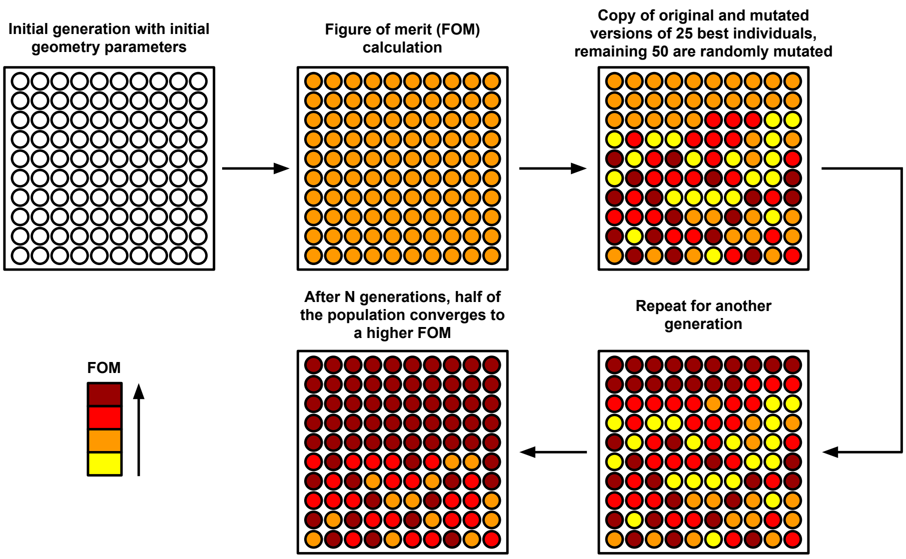


Figure 5.6: Workflow of the programmed genetic algorithm

5.2 Case study 1: MEMS capacitive accelerometer

As a first implementation of the developed program, an open-loop capacitive MEMS accelerometer is designed, simulated, and optimized. This device comprises four beams suspending a proof-mass above the substrate. The accelerometer is designed to detect acceleration in the z-axis by displacement of the proof-mass along said axis, with a mass-spring damper model illustrated in Figure 5.7.

To detect a capacitance change, the device's proof-mass is located between two electrodes with an overlap area equal to the proof-mass bottom and top surface area. The initial distance between the proof-mass and the electrodes is changed when the mass suffers a displacement caused by an acceleration, thus provoking a detectable capacitance change.

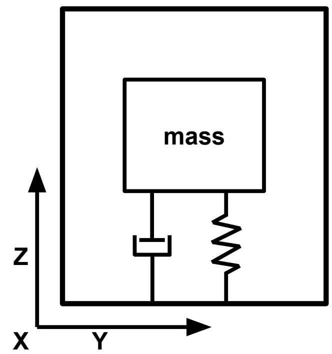


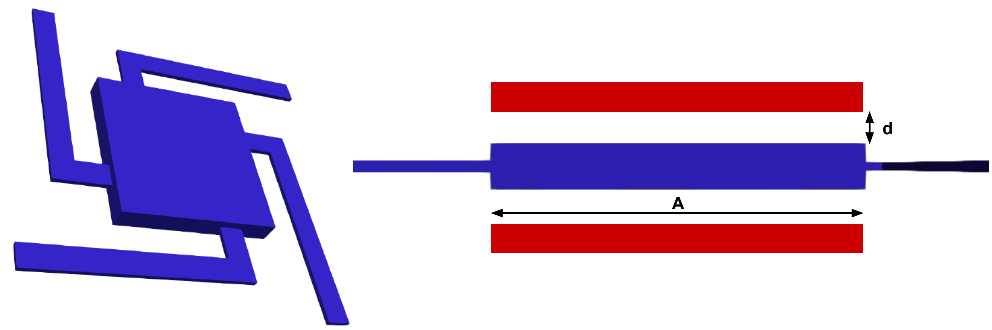
Figure 5.7: Mass-spring-damper model

5.2.1 Design analysis

The structure built by the developed software is illustrated in Figure 5.8, with its geometric parameters listed in Table 5.2. The device comprises four *L-shaped* beams connected to the proof-mass on one end, and fixed on the other. These beams suspend the proof-mass above the substrate, promoting a movement along the z-axis while restricting motion on the other directions - the mode shape corresponding to the natural frequency seen in Figure 5.9 confirms these characteristics.

The ruling capacitance change equation for this accelerometer is stated in Equation 5.14 and observed in Figure 5.8b, in which d is the distance between proof-mass and electrodes while A stands for the overlapping area of electrodes and proof-mass - given by the proof mass surface area.

In Table 5.2 the initial geometric parameters of the accelerometer are shown.



(a) 3D model generated by the software (b) Side view of device with electrodes in red

Figure 5.8: MEMS capacitive accelerometer design

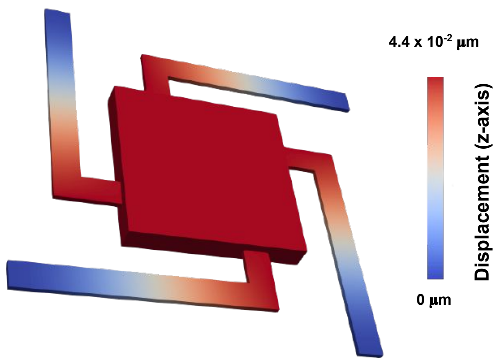


Figure 5.9: MEMS accelerometer mode shape corresponding to the natural frequency of 3284 Hz

Table 5.2: Initial geometric parameters of MEMS accelerometer

Parameter	Value
Suspension beam width	350 μm
Suspension beam length	3300 μm
Beam thickness	69 μm
Small beam length	500 μm
Proof-mass length	2400 μm
Proof-mass thickness	320 μm
Distance proof-mass/electrodes	22 μm

5.2.2 Optimization results

The process of simulation and optimization of the MEMS capacitive accelerometer is based on the system-level model observed in Figure 5.10.

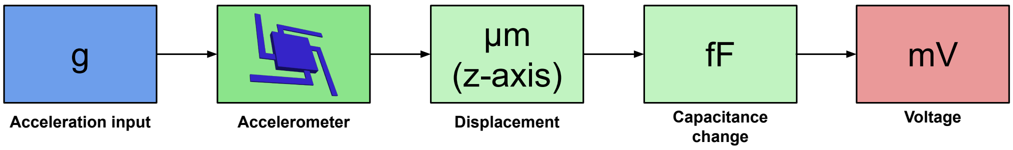


Figure 5.10: Capacitive MEMS accelerometer system-level model.

The MEMS accelerometer takes an acceleration along the z-axis as input, causing a displacement of the proof-mass that generates a detectable capacitance change, which is then read by the implemented capacitance-to-voltage circuit, governed by Equation 5.20. For the the simulated accelerometer, V_{DD} is defined as $5V$, $V_{cm} = 2.5V$, and $C_{int} = 300fF$.

The solution of a PDE is strongly related to the density of the mesh. It is, therefore, necessary to perform a mesh convergence study - in this case, the natural frequency is analyzed for simulations with different numbers of meshing elements in the suspension beams. As observed in Figure 5.11, for a number of elements of 33824 - corresponding to an element size of 55μm, the change in the first frequency mode is less than 0.15% - when compared with the next 6 points, which corresponds to a change of 30μm in element size. Thus, the remaining simulation and optimization process will consider the optimized meshing element size.

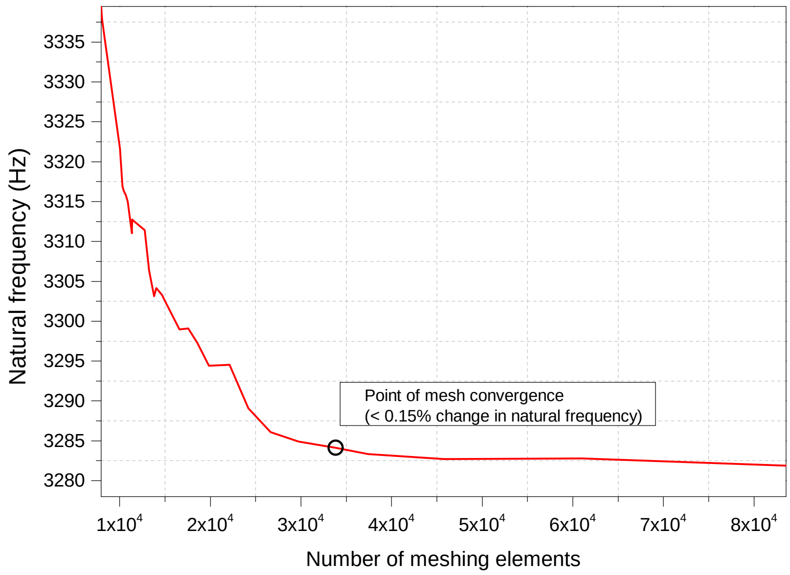


Figure 5.11: Mesh convergence study for MEMS accelerometer

The genetic algorithm optimization process described in Subsection 5.1.5 is applied to this device for 6 generations, taking into consideration a FOM defined by Equation 5.29.

$$FOM = Sensitivity(mV/g) \cdot Frequency(Hz) \cdot Q_{factor} \cdot \frac{1}{1000} \tag{5.29}$$

In this equation, *Sensitivity* stands for the output voltage when an acceleration of 1 g is applied, *Frequency* is the device's resonant frequency and Q_{factor} denotes the quality-factor of the sensing mechanism considering ambient air pressure, taking the damping into consideration.

Within 6 generations with 100 individuals each, the GA altered the chosen initial geometric parameters: proof-mass length and suspension beam width, and obtained an optimized device - the geometric changes and performance parameters are Table 5.3. The evolution of the device observed in Figure 5.12 illustrates the algorithm's tendency to reduce the suspension beam's width and to enlarge the proof-mass - this process allows for a higher sensitivity due to lower stiffness in the suspension system combined with a larger proof-mass, however, there is a decrease in the resonant frequency which limits the accelerometer bandwidth.

Table 5.3: Geometric and performance parameters of original and final accelerometer

Parameter	Original	Final	Relative change
Suspension beam width (μm)	350	152	-56.57%
Proof-mass length (μm)	2400	2613	8.15%
Sensitivity (mV/g)	80.747	237.210	193.77%
Frequency (Hz)	3284	1887	-42.54%
Q_{factor}	0.544	0.350	-35.66%
FOM	144.25	156.67	7.93%

The parameter changes observed in Figure 5.12 produced the performance parameters listed in Table 5.3.

To verify the accuracy of the modal analysis performed by the software, a comparison with COMSOL was made: the natural frequency obtained by COMSOL was 1897.15 Hz, while the natural frequency obtained by software was 1887.9 Hz - the difference was 0.5%, and so it was possible to assume the accuracy of the method.

The two drawbacks in the process were the reduction by 42.54% of the device's bandwidth and the decrease of the quality-factor by -35.66% - compensated by an increase of 193.77% in sensitivity and of 7.93% in FOM.

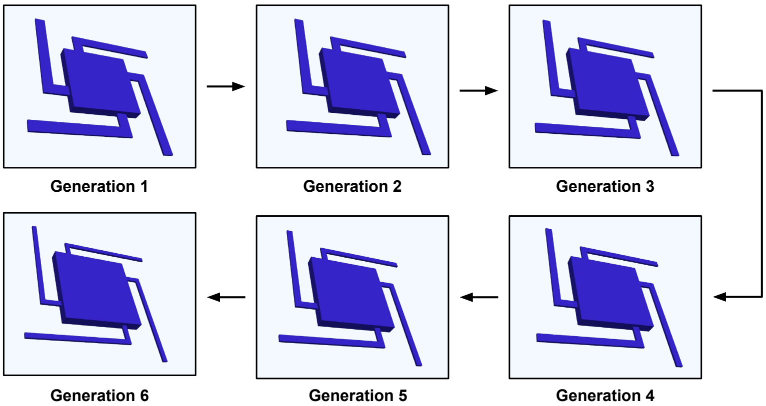


Figure 5.12: Evolution of the MEMS accelerometer through the six generations of the GA, the suspension beam width is reduced and the proof mass is enlarged

5.3 Case study 2: linear MEMS vibratory gyroscope

The second MEMS inertial sensor simulated and optimized with the developed software was a linear MEMS vibratory gyroscope, reproduced from [7]. This device featured a drive frame implemented to nest the proof-mass and thus decouple the drive and sense motion - this approach avoids the deflections in both modes present in regular linear suspension systems and it is illustrated in Figure 5.13. An *u-beam* suspension system was put together in order to ensure that both the drive and sense motion only deflect in the correct direction.

The working principle of MEMS vibratory gyroscopes was introduced in Subsection 3.1.2: these devices maintain a drive oscillation that allows for the detection of the Coriolis force generated by an angular rate input - the force will result in an energy transfer from the drive axis to the sense axis which occurs in the form of a proof-mass movement along said axis.

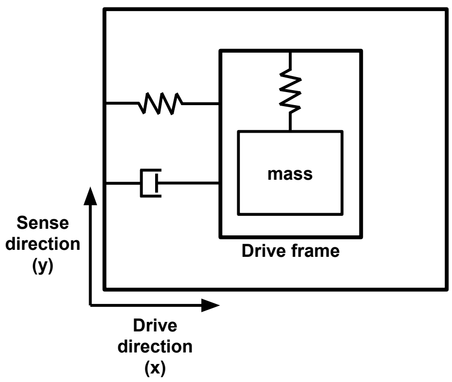
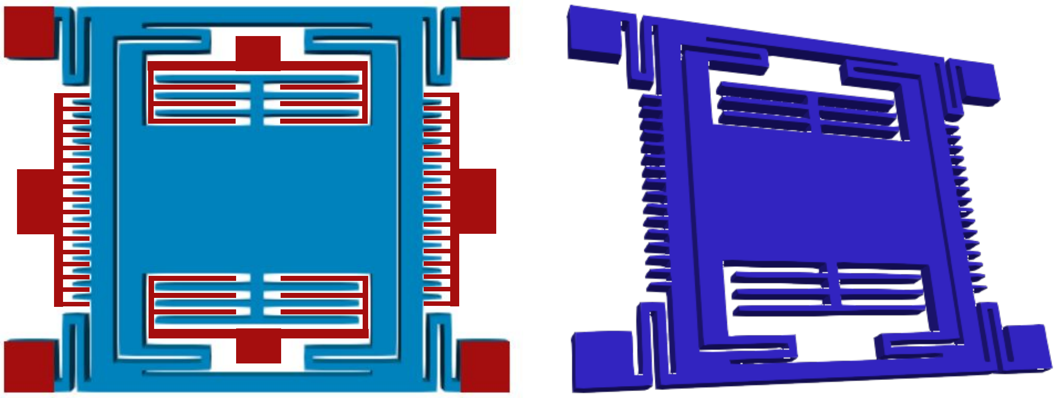


Figure 5.13: Mass-spring-damper model of MEMS gyroscope design

5.3.1 Design analysis

The geometry built by the software is shown in Figure 5.14, with initial geometric parameters listed in Table 5.4. This design comprises eight anchors: four of them anchoring the suspension *u-beams* connected to the drive frame, two stationary electrodes for sensing, and two stationary drive electrodes.

The *u-beams* are designed to perform as a suspension system that keep the proof-mass above the substrate while eliminating nonlinearity and axial-loading limitations present in the simple single fixed beams. The four drive frame beams promote movement along the x-axis - the drive direction, and the four beams that connect the proof-mass to the drive frame facilitate a displacement by the y-axis - the sense direction.



(a) MEMS gyroscope design, with stationary parts in red (b) 3D model generated by the software

Figure 5.14: Linear vibratory MEMS gyroscope design [7]

Table 5.4: Initial geometric parameters of MEMS gyroscope

Parameter	Value
Suspension beam width	20 μm
Suspension beam length	194 μm
Drive frame length	970 μm
Proof-mass lateral beam width	60 μm
Proof-mass lateral beam length	430 μm
Proof-mass width	580 μm
Proof-mass length	440 μm
Comb finger width	14 μm
Drive comb finger length	48 μm
Sense comb finger length	243 μm
Thickness	50 μm

In this gyroscope, drive oscillation along the x-axis was possible due to balanced variable-area electrostatic actuation, which was done through the lateral electrodes seen in Figure 5.14a. The sensing principle applied in the device was the differential sensing, made possible by two sets of variable-gap comb-fingers observed inside the drive frame.

5.3.2 Optimization results

The simulation and optimization of the linear vibratory MEMS gyroscope is based on the system-level model illustrated in Figure 5.15.

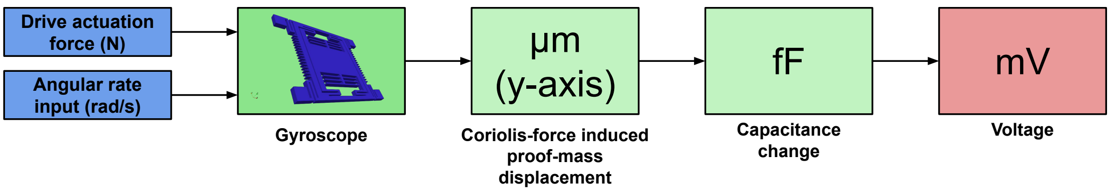


Figure 5.15: Linear vibratory MEMS gyroscope system-level model.

The device takes a drive actuation force governed by Equation 5.19 and generated by the aforementioned driving electrodes, as well as an angular-rate around the z-axis as input. The software takes the drive actuation force and simulates its effect on the MEMS structure, taking into consideration damping (in this mechanism, slide-film damping is the most prominent damping factor), and obtaining the resulting drive amplitude. For this actuation mechanism, a DC voltage of $8V$ and an AC voltage of $4V$ are applied.

The gyroscope then makes use of the driving velocity along the x-axis and the angular rate input to generate a Coriolis force along the y-axis, providing a displacement of the proof-mass. This displacement causes a detectable capacitance change, described by Equation 5.17, which is then read by the implemented capacitance-to-voltage circuit, governed by Equation 5.20. For the the simulated gyroscope, V_{DD} is defined as $5V$, $V_{cm} = 2.5V$, and $C_{int} = 100fF$.

For this gyroscope, the first frequency mode is analyzed for simulations with different numbers of meshing elements. As observed in Figure 5.16, for a number of elements of 9737 - corresponding to an element size of 80μm, the change in the first frequency mode is less than 0.15% - when compared with the next 6 points, which corresponds to a change of 30μm in element size. Thus, the remaining simulation and optimization process will consider the optimized meshing element size.

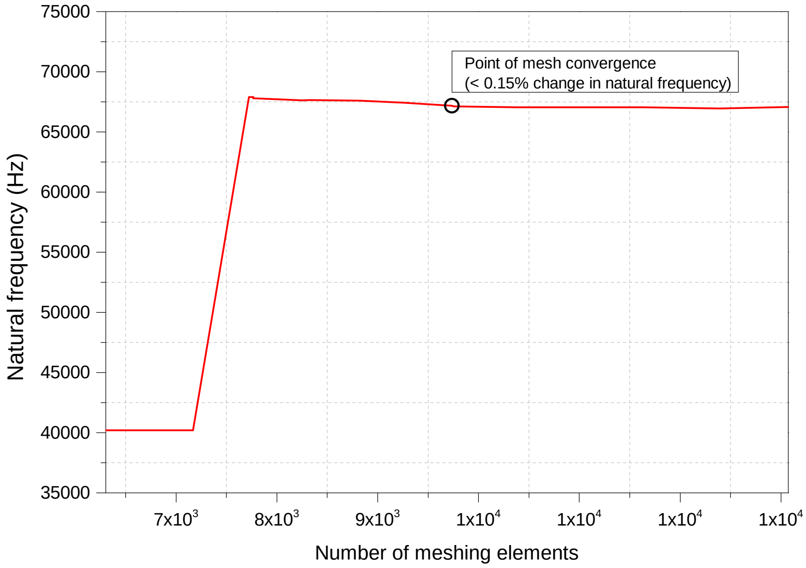


Figure 5.16: Mesh convergence study for MEMS gyroscope

The genetic algorithm optimization process described in Subsection 5.1.5 is applied to this device for 6 generations. The FOM considered by the GA is defined by Equation 5.30.

$$FOM = (Sensitivity(mV/rads^{-1}) \cdot \frac{1}{\Delta f} \cdot Q_{sense}) \cdot 10^6 \tag{5.30}$$

In this equation, $Sensitivity$ is given by the output voltage when the device is subjected to an angular rate of $1 rads^{-1}$ around the z-axis, Δf denotes the difference between drive and sense frequency - this parameter is of importance to achieve maximum mechanical gain in the sense mode concerning the input angular rate. Looking at Equation 5.31 [7], it becomes clear that it is desirable to match drive and sense resonant frequencies. Lastly, Q_{sense} represents the sense mode quality factor.

$$y_0 = \Omega_z \frac{m_C \omega_D}{m_S \omega_S^2} \frac{2x_0}{\sqrt{[1-(\frac{\omega_D}{\omega_S})^2] + [\frac{1}{Q_{sense}} \frac{\omega_D}{\omega_S}]^2}} \tag{5.31}$$

During the 6 generations with 100 individuals, the algorithm altered the chosen initial geometric parameters to find the optimal device - the geometric changes and performance parameters are listed in Table 5.5. The chosen parameters were: suspension beam's width, proof-mass width and length, proof-mass frame width and length, and sense comb fingers width. The evolution of the optimization is illustrated in Figure 5.17, in which the best device from each generation is displayed.

The suspension beam's width saw a severe reduction (similarly to the accelerometer's springs), on the other hand, the optimized proof-mass became larger than the original - this combination leads to an increase in compliancy of the whole structure, and thus, in sensitivity. Moreover, the drive and sense mode frequencies as well as the frequency split were significantly reduced. A comparison between frequency modes of the original and optimized device is shown in Figure 5.18.

Table 5.5: Geometric and performance parameters of original and optimized gyroscope

Parameter	Original	Final	Relative change
Suspension beam width (μm)	20	9	-55.00%
Proof-mass frame width (μm)	430	395	-8.14%
Proof-mass frame length (μm)	60	54	-10.00%
Proof-mass width (μm)	290	309	6.15%
Proof-mass length (μm)	220	240	8.33%
Sense finger width (μm)	14	9	-35.71%
Sensitivity (mV/rads^{-1})	1.546	8.054	420.9%
Δf	13296	5792	-56.44%
Q_{sense}	1.917	0.952	-50.34%
FOM	222.9	1323	493.5%

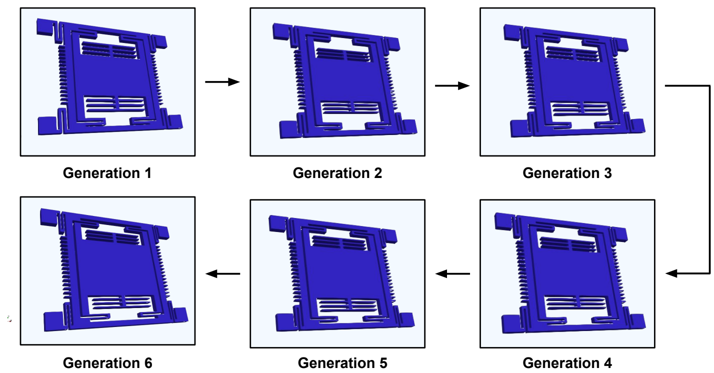


Figure 5.17: Evolution of the MEMS gyroscope through the six generations of the GA - the proof-mass became larger; the u-beams, the sense comb fingers, and the proof-mass frame became thinner

The frequency mode shapes illustrated in Figure 5.18 represent the magnitude of movement in the x-axis for a driving motion, or in the y-axis for a sensing motion. It becomes visible that in the optimized device, the undesired y-axis movement in the drive frame is reduced, while the desired proof-mass displacement is slightly enhanced.

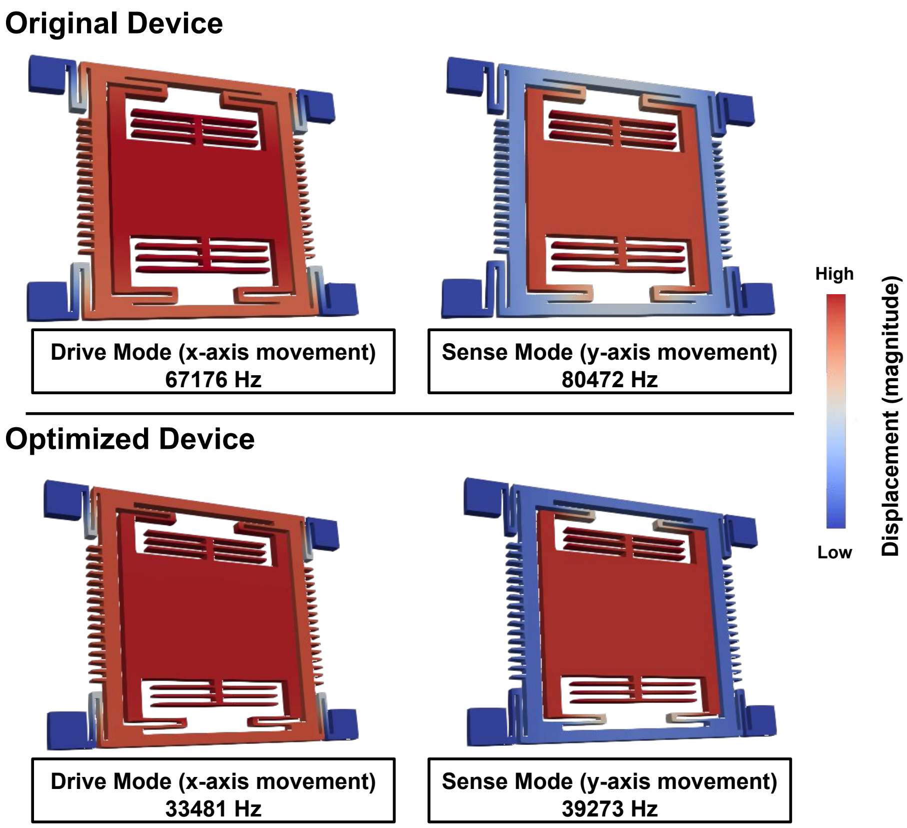


Figure 5.18: Frequency modes of original and optimized MEMS gyroscope, the movement modes became more pronounced

The changes observed in Figure 5.17 produced the performance changes listed in Table 5.5. The only drawback of the optimization process was the decrease of 50.34% in the sense mode's quality factor - it was largely compensated by a 420.9% increase in sensitivity and by a decrease of 56.44% in frequency mismatch. Overall, the FOM improved by 493.5%.

Chapter 6

Conclusion and Future Perspectives

The advent of micromachining technology brought a never-ending range of possible applications for microsized sensors. In the inertial sensor's area, the experienced growth opens the door to numerous improvements in quality of life as well as life-saving applications in the automotive industry. This array of new technologies in fabrication and product possibilities would benefit from a similar development in the process of design, simulation and optimization.

This study presented a novel electro-mechanical co-optimization methodology for MEMS inertial sensors, entirely based on Python. A software comprising geometry design, a finite element method simulation, damping calculation, electronic domain simulation, and a genetic algorithm optimization process - was developed and applied to two MEMS inertial sensors.

The software can build geometries with relative complexity, making use of the *Pygmsh* Python library. Although the most complex geometric operations such as filet, chamfer, bezier curve, and arrays are not available; the vast majority of MEMS inertial sensors can be built by code within the software.

The geometry building block passes its parameters and mesh to the FEM part of the program, which is able to process modal analysis and displacement accurately - with a reported 0.5% difference to the frequency modes obtained with COMSOL.

The damping script takes into account the geometric parameters and the results from the modal analysis, calculating the squeeze film and slide film damping coefficient, and later the quality factor.

Depending on what type of actuation or sensing the device has, the electronic block of the software calculates the capacitance change, passing the values to the implemented circuit which in turn computes the voltage output of the system.

Finally, the implemented genetic algorithm takes the performance parameters of the devices and proceeds to select the best individuals out of each generation, achieving a co-optimized sensor in the end. To validate the software, two MEMS inertial sensors were designed, simulated and optimized.

The first optimized device was an open-loop capacitive MEMS accelerometer:

the co-optimization process increased the sensitivity by 193.77% and the FOM by 7.93%, compensating for the loss of 42.54% in resonant frequency and of 35.66%. The first version of this implementation resulted in a publication titled "Electro-mechanical Co-Optimization of MEMS Devices in Python" - submitted, accepted, and presented at IEEE SENSORS 2020 conference.

The second optimized device was an open-loop Coriolis vibratory MEMS gyroscope: the co-optimization process improved the sensitivity by 420.9% and the frequency mismatch decreased by 56.44%, compensating the 50.34% loss in the quality-factor and producing an overall improvement in FOM of 493.5%. The co-optimization of this device resulted in a publication titled "Python-based Electro-mechanical Co-optimization of MEMS Inertial Sensors" - submitted to IEEE MEMS21 conference.

The developed Python solution is a powerful tool that provides designers with limitless customization freedom, presenting engineers with complete control of all steps in simulation and optimization - allowing an efficient management of computational resources according to specific research goals.

In order for this software to become of professional-grade, it is necessary to implement transient and closed-loop system simulation and optimization. Also, a non-linearity calculation and integration into the software is essential if the goal is to simulate the real performance of the devices

Bibliography

[1] D. K. Shaeffer. "MEMS inertial sensors: A tutorial overview." In: *IEEE Communications Magazine* 51.4 (2013), pp. 100–109.

[2] C. Acar, A. R. Schofield, A. A. Trusov, L. E. Costlow, and A. M. Shkel. "Environmentally Robust MEMS Vibratory Gyroscopes for Automotive Applications." In: *IEEE Sensors Journal* 9.12 (2009), pp. 1895–1906.

[3] Y. Dong. "8 - MEMS inertial navigation systems for aircraft." In: *MEMS for Automotive and Aerospace Applications*. Ed. by M. Kraft and N. M. White. Woodhead Publishing Series in Electronic and Optical Materials. Woodhead Publishing, 2013, pp. 177–219. ISBN: 978-0-85709-118-5. DOI: https://doi.org/10.1533/9780857096487.2.177. URL: http://www.sciencedirect.com/science/article/pii/B9780857091185500084.

[4] G. Bhatt, K. Manoharan, P. S. Chauhan, and S. Bhattacharya. "MEMS Sensors for Automotive Applications: A Review." In: *Sensors for Automotive and Aerospace Applications*. Ed. by S. Bhattacharya, A. K. Agarwal, O. Prakash, and S. Singh. Singapore: Springer Singapore, 2019, pp. 223–239. ISBN: 978-981-13-3290-6. DOI: 10.1007/978-981-13-3290-6_12. URL: https://doi.org/10.1007/978-981-13-3290-6_12.

[5] B. A. Abruzzo and T. G. Recchia. "Online Calibration of Inertial Sensors for Range Correction of Spinning Projectiles." In: *Journal of Guidance, Control, and Dynamics* 39.8 (2016), pp. 1918–1924. DOI: 10.2514/1.G001809. eprint: https://doi.org/10.2514/1.G001809. URL: https://doi.org/10.2514/1.G001809.

[6] J. S. Han, E. B. Rudnyi, and J. G. Korvink. "Efficient optimization of transient dynamic problems in MEMS devices using model order reduction." In: *Journal of Micromechanics and Microengineering* 15.4 (Mar. 2005), pp. 822–832. DOI: 10.1088/0960-1317/15/4/021. URL: https://doi.org/10.1088%2F0960-1317%2F15%2F4%2F021.

[7] C. Acar and A. Shkel. *MEMS vibratory gyroscopes: structural approaches to improve robustness*. Springer Science & Business Media, 2008.

[8] T. Masuzawa. "State of the art of micromachining." In: *Cirp Annals* 49.2 (2000), pp. 473–488.

[9] S. Beeby, G. Ensel, N. White, and M. Kraft. *MEMS Mechanical Sensors*. Artech House MEMS Library. Artech House, 2004. ISBN: 9781580535366. URL: https://books.google.be/books?id=i0UwDwAAQBAJ.

[10] Guan Xin, Yan Dong, and Gao Zhen-hai. "Study on errors compensation of a vehicular MEMS accelerometer." In: *IEEE International Conference on Vehicular Electronics and Safety, 2005.* 2005, pp. 205–210. DOI: 10.1109/ICVES.2005.1563642.

[11] S. D. Senturia. "Perspectives on MEMS, past and future: the tortuous pathway from bright ideas to real products." In: *TRANSDUCERS '03. 12th International Conference on Solid-State Sensors, Actuators and Microsystems. Digest of Technical Papers (Cat. No.03TH8664)*. Vol. 1. 2003, 10–15 vol.1. DOI: 10.1109/SENSOR.2003.1215241.

[12] D. Glassbrenner and M. Starnes. *Lives saved calculations for seat belts and frontal air bags*. Washington, DC: National Highway Traffic Safety Administration, 2009.

[13] J. Marek. "MEMS for automotive and consumer electronics." In: *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE. 2010, pp. 9–17.

[14] S. Kusmakar, C. K. Karmakar, B. Yan, T. J.O'Brien, R. Muthuganapathy, and M. Palaniswami. "Automated Detection of Convulsive Seizures Using a Wearable Accelerometer Device." In: *IEEE Transactions on Biomedical Engineering* 66.2 (2019), pp. 421–432. DOI: 10.1109/TBME.2018.2845865.

[15] A. Sucerquia, J. D. López, and J. F. Vargas-Bonilla. "Real-life/real-time elderly fall detection with a triaxial accelerometer." In: *Sensors* 18.4 (2018), p. 1101.

[16] S. B. Khojasteh, J. R. Villar, C. Chira, V. M. González, and E. De la Cal. "Improving fall detection using an on-wrist wearable accelerometer." In: *Sensors* 18.5 (2018), p. 1350.

[17] P. Van Thanh, D.-T. Tran, D.-C. Nguyen, N. D. Anh, D. N. Dinh, S. El-Rabaie, and K. Sandrasegaran. "Development of a real-time, simple and high-accuracy fall detection system for elderly using 3-DOF accelerometers." In: *Arabian Journal for Science and Engineering* 44.4 (2019), pp. 3329–3342.

[18] M. Dencker and L. B. Andersen. "Accelerometer-measured daily physical activity related to aerobic fitness in children and adolescents." In: *Journal of sports sciences* 29.9 (2011), pp. 887–895.

[19] Guangchun Li, Yunfeng He, Yanhui Wei, Shenbo Zhu, and Yanzhe Cao. "The MEMS gyro stabilized platform design based on Kalman Filter." In: *2013 International Conference on Optoelectronics and Microelectronics (ICOM)*. 2013, pp. 14–17. DOI: 10.1109/ICoOM.2013.6626480.

[20] V. Kempe. *Inertial MEMS: Principles and Practice*. Cambridge University Press, 2011, pp. 14–24. ISBN: 9781139494823. URL: https://books.google.be/books?id=XzdvDGbLZ8EC.

[21] M. Armenise, C. Ciminelli, F. Dell'Olio, and V. Passaro. *Advances in Gyroscope Technologies*. Springer Berlin Heidelberg, 2010. ISBN: 9783642154942. URL: https://books.google.be/books?id=lJUiyigJRBgC.

[22] W. Somerville. "The description of Foucault's pendulum." In: *Quarterly Journal of the Royal Astronomical Society* 13 (1972), pp. 40–62.

[23] A. Persson. "The Coriolis Effect–a conflict between common sense and mathematics." In: *Italian Meteorological Society* (2005), pp. 20–31.

[24] V. Apostolyuk. *Coriolis Vibratory Gyroscopes: Theory and Design*. Springer International Publishing, 2015. ISBN: 9783319221984. URL: https://books.google.be/books?id=hBRcCgAAQBAJ.

[25] C. Darwin. *The origin of species*. PF Collier & son New York, 1909.

[26] G. Mendel. *Experiments in plant hybridisation*. Harvard University Press, 1965.

[27] D. S. Weile and E. Michielssen. "Genetic algorithm optimization applied to electromagnetics: A review." In: *IEEE Transactions on Antennas and Propagation* 45.3 (1997), pp. 343–353.

[28] Y. Zhang, R. Kamalian, A. M. Agogino, and C. H. Séquin. "Hierarchical MEMS synthesis and optimization." In: *Smart Structures and Materials 2005: Smart Electronics, MEMS, BioMEMS, and Nanotechnology*. Vol. 5763. International Society for Optics and Photonics. 2005, pp. 96–106.

[29] J. S. Han, E. B. Rudnyi, and J. G. Korvink. "Efficient optimization of transient dynamic problems in MEMS devices using model order reduction." In: *Journal of Micromechanics and Microengineering* 15.4 (2005), p. 822.

[30] E. S. Hung, Y.-J. Yang, and S. D. Senturia. "Low-order models for fast dynamical simulation of MEMS microstructures." In: *Proceedings of International Solid State Sensors and Actuators Conference (Transducers' 97)*. Vol. 2. IEEE. 1997, pp. 1101–1104.

[31] M. Kudryavtsev, E. B. Rudnyi, J. G. Korvink, D. Hohlfeld, and T. Bechtold. "Computationally efficient and stable order reduction methods for a large-scale model of MEMS piezoelectric energy harvester." In: *Microelectronics Reliability* 55.5 (2015), pp. 747–757.

[32] A. H. Nayfeh, M. I. Younis, and E. M. Abdel-Rahman. "Reduced-order models for MEMS applications." In: *Nonlinear dynamics* 41.1-3 (2005), pp. 211–236.

[33] C. Wang, H. Liu, X. Song, F. Chen, I. Zeimpekis, Y. Wang, J. Bai, K. Wang, and M. Kraft. "Genetic algorithm for the design of freeform geometries in a MEMS accelerometer comprising a mechanical motion pre-amplifier." In: *2019 20th International Conference on Solid-State Sensors, Actuators and Microsystems & Eurosensors XXXIII (TRANSDUCERS & EUROSENSORS XXXIII)*. IEEE. 2019, pp. 2099–2102.

[34] S. COMSOL AB Stockholm. *COMSOL Multiphysics®*. Version 5.4. URL: www.comsol.com.

[35] I. The Mathworks. *MATLAB:2017b*. Natick, Massachusetts, 2017.

[36] M. R. Solouk, M. H. Shojaeefard, and M. Dahmardeh. "Parametric topology optimization of a MEMS gyroscope for automotive applications." In: *Mechanical Systems and Signal Processing* 128 (2019), pp. 389–404. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2019.03.049.

[37] J. T. Oden. "Finite Elements : Introduction." In: *Handbook of Numerical Analysis Volime II : Finite Element Methods (Part1)* (1991), pp. 3–12.

[38] J. Fritz. *Partial Differential Equations*. New York: Springer-Verlag, 1982. ISBN: 0-387-90609-6.

[39] A. Logg, K.-A. Mardal, G. N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by A. Logg, K.-A. Mardal, and G. N. Wells. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

[40] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[41] K. J. Millman and M. Aivazis. "Python for Scientists and Engineers." In: *Computing in Science Engineering* 13.2 (2011), pp. 9–12.

[42] T. E. Oliphant. "Python for Scientific Computing." In: *Computing in Science Engineering* 9.3 (2007), pp. 10–20.

[43] N. Schlömer, A. Cervone, G. McBain, Tryfon-Mw, R. V. Staden, F. Gokstorp, Toothstone, J. S. Dokken, Anzil, and J. Sanchez. *nschloe/pygmsh v6.1.1*. 2020. DOI: 10.5281/zenodo.3764683.

[44] C. Geuzaine and J.-F. Remacle. "Gmsh: a three-dimensional 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.

[45] N. Schlömer, G. McBain, K. Luu, christos, T. Li, V. M. Ferrándiz, C. Barnes, L. Dalcin, eolianoe, and nilswagner. *nschloe/meshio v4.0.12*. 2020. DOI: 10.5281/zenodo.3773318.

[46] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. "The FEniCS Project Version 1.5." In: *Archive of Numerical Software* 3.100 (2015). DOI: 10.11588/ans.2015.100.20553.

[47] A. Logg and G. N. Wells. "DOLFIN: Automated Finite Element Computing." In: *ACM Transactions on Mathematical Software* 37.2 (2010). DOI: 10.1145/1731022.1731030.

[48] A. Lurie and A. Belyaev. *Theory of Elasticity*. Foundations of Engineering Mechanics. Springer Berlin Heidelberg, 2010, pp. 151–155. ISBN: 9783540264552. URL: https://books.google.be/books?id=saEqz%5C_LKkdEC.

[49] J. Brewer. "Divergence of a vector field." In: *Vector Calculus* 7 (1999).

[50] D. R. Smith. "The Cauchy Stress Tensor." In: *An Introduction to Continuum Mechanics — after Truesdell and Noll*. Dordrecht: Springer Netherlands, 1993, pp. 143–162. ISBN: 978-94-017-0713-8. DOI: 10.1007/978-94-017-0713-8_5. URL: https://doi.org/10.1007/978-94-017-0713-8_5.

[51] Z.-C. S. K. Feng. *Mathematical Theory of Elastic Structures*. Springer New York, 1981, pp. 106–108. ISBN: 0-387-51326-4.

[52] A. H.-D. Cheng and D. T. Cheng. "Heritage and early history of the boundary element method." In: *Engineering Analysis with Boundary Elements* 29.3 (2005), pp. 268–302.

[53] J. Wortman and R. Evans. "Young's modulus, shear modulus, and Poisson's ratio in silicon and germanium." In: *Journal of applied physics* 36.1 (1965), pp. 153–156.

[54] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[55] R. Clough and J. Penzien. *Dynamics of Structures*. McGraw-Hill, 1993, p. 201. ISBN: 9780071132411. URL: https://books.google.be/books?id=HxLakQEACAAJ.

[56] U. Ayachit. *The ParaView Guide: A Parallel Visualization Applications*. Kitware, 2015. ISBN: 978-1930934306.

[57] R. F. Harrington. *Introduction to electromagnetic engineering*. Courier Corporation, 2003.

[58] R. W. Johnstone and M. Parameswaran. "Electrostatic Actuators." In: *An Introduction to Surface-Micromachining*. Springer, 2004, pp. 135–152.

[59] A. Utz, C. Walk, N. Haas, T. Fedtschenko, A. Stanitzki, M. Mokhtari, M. Görtz, M. Kraft, and R. Kokozinski. "An ultra-low noise capacitance to voltage converter for sensor applications in 0.35 μm CMOS." In: *Journal of Sensors and Sensor Systems* 6.2 (2017), pp. 285–301.

[60] M. Bao and H. Yang. "Squeeze film air damping in MEMS." In: *Sensors and Actuators A: Physical* 136.1 (2007), pp. 3–27.

[61] Springer Verlag GmbH, European Mathematical Society. *Encyclopedia of Mathematics: Knudsen Number*. Website. URL: http://encyclopediaofmath.org/index.php?title=Knudsen_number&oldid=13592.

[62] T. Veijola, H. Kuisma, J. Lahdenperä, and T. Ryhänen. "Equivalent-circuit model of the squeezed gas film in a silicon accelerometer." In: *Sensors and Actuators A: Physical* 48.3 (1995), pp. 239–248.

[63] J. J. Blech. "On isothermal squeeze films." In: (1983).

[64] T. Veijola and M. Turowski. "Compact damping models for laterally moving microstructures with gas-rarefaction effects." In: *Journal of Microelectromechanical Systems* 10.2 (2001), pp. 263–273.

[65] W. Siebert. *Circuits, Signals, and Systems*. MIT electrical engineering and computer science series. McGraw-Hill, 1986. ISBN: 9780262192293. URL: https://books.google.be/books?id=zBTUiIrb2WIC.

[66] D. Alciatore and M. Histand. *Introduction to Mechatronics and Measurement Systems*. Engineering Series. McGraw-Hill Companies,Incorporated, 2007. ISBN: 9780072963052. URL: https://books.google.be/books?id=V0FSAAAAMAAJ.

Annex I

Software implementation on MEMS accelerometer

The developed software code, implemented on a MEMS accelerometer, is displayed on the following listings. Several software packages and Python libraries are necessary in order to run the program:

- Software packages: *Gmsh*
- Python libraries: *Pygmsh, meshio, FEniCS, math, numpy, random, copy*

The user should build the initial device in the geometry building script and adapt the following simulation and optimization scripts to the geometric parameters of said device - finally the program is ran from the genetic algorithm script.

Listing I.1: Geometry building block

```
# MEMS Accelerometer 3D Geometry
# @ruiesteves

# Imports
import pygmsh as pg # geometry & meshing definition
import meshio # meshing export



# Functions
def build(suspension_beam_width,proof_mass_length):

    # Geometric parameters
    cl = 55
    proof_mass_cl = 150
    scale = 1e-6
    suspension_beam_length = 3300*scale
    beam_thickness = 69*scale
    small_beam_length = 500*scale
    proof_mass_thickness = 320*scale
    beam_dist = 500*scale
    beam_l = 122.5 * scale
```

```
beam_h = 177.5 * scale
beam_dist2 = beam_dist + suspension_beam_length
beam_dist3 = proof_mass_length + suspension_beam_width + small_beam_length
beam_dist_final = beam_dist2 - beam_dist3
beam_lower = (proof_mass_thickness - beam_thickness)/2
beam_to_mass = (beam_dist_final+suspension_beam_width+small_beam_length +
    ↪ proof_mass_length) - suspension_beam_length

# Geometry build
geom = pg.opencascade.Geometry()

# Beam1_1
p1 = [0,beam_dist_final,beam_lower]
p2 = [suspension_beam_length,suspension_beam_width,beam_thickness]
beam1_1 = geom.add_box(p1,p2,char_length=cl*scale)

# Beam1_2
p3 = [suspension_beam_length-suspension_beam_width,beam_dist_final +
    ↪ suspension_beam_width,beam_lower]
p4 = [suspension_beam_width,small_beam_length,beam_thickness]
beam1_2 = geom.add_box(p3,p4,char_length=cl*scale)

# Beam1 complete
beam1 = geom.boolean_union([beam1_1,beam1_2])

# Proof mass
p1 = [beam_dist_final+suspension_beam_width+small_beam_length,
    ↪ beam_dist_final+suspension_beam_width+small_beam_length,0]
p2 = [proof_mass_length,proof_mass_length,proof_mass_thickness]
proof_mass = geom.add_box(p1,p2,char_length=proof_mass_cl*scale)

# Beam2_1
p1 = [beam_dist_final,beam_dist_final+suspension_beam_width+
    ↪ small_beam_length+beam_to_mass,beam_lower]
p2 = [suspension_beam_width,suspension_beam_length,beam_thickness]
beam2_1 = geom.add_box(p1,p2,char_length=cl*scale)

# Beam2_2
p1 = [beam_dist_final+suspension_beam_width,beam_dist_final+
    ↪ suspension_beam_width+small_beam_length+beam_to_mass,beam_lower]
p2 = [small_beam_length,suspension_beam_width,beam_thickness]
beam2_2 = geom.add_box(p1,p2,char_length=cl*scale)

# Beam 2
beam2 = geom.boolean_union([beam2_1,beam2_2])
```

```
    # Beam3_1
    p1 = [beam_dist_final+suspension_beam_width+small_beam_length+proof_mass_length,beam_dist_final+suspension_beam_width+small_beam_length+proof_mass_length - beam_to_mass - suspension_beam_width,beam_lower]
    p2 = [small_beam_length,suspension_beam_width,beam_thickness]
    beam3_1 = geom.add_box(p1,p2,char_length=cl*scale)

    # Beam3_2
    p1 = [beam_dist_final+suspension_beam_width+small_beam_length+proof_mass_length+small_beam_length,0,beam_lower]
    p2 = [suspension_beam_width,suspension_beam_length,beam_thickness]
    beam3_2 = geom.add_box(p1,p2,char_length=cl*scale)

    # Beam3
    beam3 = geom.boolean_union([beam3_1,beam3_2])


    # Beam4_1
    p1 = [beam_dist_final+suspension_beam_width+small_beam_length+beam_to_mass,beam_dist_final+suspension_beam_width+small_beam_length+proof_mass_length,beam_lower]
    p2 = [suspension_beam_width,small_beam_length,beam_thickness]
    beam4_1 = geom.add_box(p1,p2,char_length=cl*scale)

    # Beam4_2
    p1 = [beam_dist_final+suspension_beam_width+small_beam_length+beam_to_mass,beam_dist_final+suspension_beam_width+small_beam_length+proof_mass_length+small_beam_length,beam_lower]
    p2 = [suspension_beam_length,suspension_beam_width,beam_thickness]
    beam4_2 = geom.add_box(p1,p2,char_length=cl*scale)

    # Beam 4
    beam4 = geom.boolean_union([beam4_1,beam4_2])


    #Complete Union
    final = geom.boolean_union([beam1,proof_mass,beam2,beam3,beam4])

    mesh = pg.generate_mesh(geom,gmsh_path="/home/ruiesteves/Documents/Tese/MechanicalModel/gmsh-4.5.2-Linux64/bin/gmsh") # Be sure to change the gmsh_path to the installed folder
    meshio.write("accelerometer.xml",mesh)
```

Listing I.2: FEM displacement script

```
# MEMS accelerometer displacement simulation script
# @ruiesteves

# Imports
from __future__ import print_function
from dolfin import *

# Material constants
E = Constant(170e9)
nu = Constant(0.28)
rho = 2329
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)

# Mesh
mesh = Mesh('accelerometer.xml')


def disp(suspension_beam_width,proof_mass_length):

    # Constants
    scale = 1e-6
    suspension_beam_length = 3300*scale
    beam_thickness = 69*scale
    small_beam_length = 500*scale
    proof_mass_thickness = 320*scale
    beam_dist = 500*scale
    beam_l = 122.5 * scale
    beam_h = 177.5 * scale
    beam_dist2 = beam_dist + suspension_beam_length
    beam_dist3 = proof_mass_length + suspension_beam_width + small_beam_length
    beam_dist_final = beam_dist2 - beam_dist3
    beam_lower = (proof_mass_thickness - beam_thickness)/2
    beam_to_mass = (beam_dist_final+suspension_beam_width+small_beam_length +
        ↪ proof_mass_length) - suspension_beam_length
    anchor_top = beam_dist_final+suspension_beam_width+small_beam_length+
        ↪ beam_to_mass+suspension_beam_length
    volume_PM = proof_mass_length**2 * proof_mass_thickness

    # Strain operator
    def eps(v):
        return sym(grad(v))

    # Stress tensor
```

```
def sigma(v):
    return lmbda*tr(eps(v))*Identity(3) + 2.0*mu*eps(v)


# Boundary
def left(x,on_boundary):
    return near(x[0],0.)

def bottom(x,on_boundary):
    return near(x[1],0.)

def top(x,on_boundary):
    return near(x[1],anchor_top)

def right(x,on_boundary):
    return near(x[0],anchor_top)

def main():
    rho_g = 9.8*(rho)
    print("Volume:",volume_PM)
    print("Force:",rho_g)
    f = Constant((0.,0.,rho_g))
    T = Constant((0,0,0))
    V = VectorFunctionSpace(mesh,'Lagrange',degree=3)
    du = TrialFunction(V)
    u_ = TestFunction(V)
    a = inner(sigma(du),eps(u_))*dx
    l = dot(f,u_)*dx

    bc = [DirichletBC(V, Constant((0.,0.,0.)),left),
    DirichletBC(V, Constant((0.,0.,0.)),right),
    DirichletBC(V, Constant((0.,0.,0.)),top),
    DirichletBC(V, Constant((0.,0.,0.)),bottom)]
    u = Function(V, name='Displacement')
    solve(a == l, u, bc)
    z_disp = u(beam_dist_final+suspension_beam_width+small_beam_length+proof_mass_length/2,beam_dist_final+suspension_beam_width+small_beam_length+proof_mass_length/2,proof_mass_thickness/2)

    # Set up file for exporting results
    file_results = XDMFFile("acc_displacement.xdmf")
    file_results.parameters["flush_output"] = True
    file_results.parameters["functions_share_mesh"] = True
    file_results.write(u,0)

```

```
        print(z_disp[2]*1e6,"um")
        return z_disp[2]

    return main()
```

Listing I.3: Modal analysis simulation script

```
# MEMS accelerometer modal analysis script
# @ruiesteves

# Imports
from fenics import *
import numpy as np

# Material constants
E = Constant(170e9)
nu = Constant(0.28)
rho = 2329
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)

# Meshing
mesh = Mesh('accelerometer.xml')


def main(suspension_beam_width,proof_mass_length):

    # Constants
    cl = 175
    proof_mass_cl = 150
    scale = 1e-6
    suspension_beam_length = 3300*scale
    beam_thickness = 69*scale
    small_beam_length = 500*scale
    proof_mass_thickness = 320*scale
    beam_dist = 500*scale
    beam_l = 122.5 * scale
    beam_h = 177.5 * scale
    beam_dist2 = beam_dist + suspension_beam_length
    beam_dist3 = proof_mass_length + suspension_beam_width + small_beam_length
    beam_dist_final = beam_dist2 - beam_dist3
    beam_lower = (proof_mass_thickness - beam_thickness)/2
    beam_to_mass = (beam_dist_final+suspension_beam_width+small_beam_length +
        ↪ proof_mass_length) - suspension_beam_length
    anchor_top = beam_dist_final+suspension_beam_width+small_beam_length+
        ↪ beam_to_mass+suspension_beam_length
```

```

# Functions
def eps(v):
    return sym(grad(v))

def sigma(v):
    return lmbda*tr(eps(v))*Identity(3) + 2.0*mu*eps(v)

# Function Space
V = VectorFunctionSpace(mesh,'Lagrange',degree=3)
u_ = TrialFunction(V)
du = TestFunction(V)

# Boundary
def left(x,on_boundary):
    return near(x[0],0.)

def bottom(x,on_boundary):
    return near(x[1],0.)

def top(x,on_boundary):
    return near(x[1],anchor_top)

def right(x,on_boundary):
    return near(x[0],anchor_top)

bc = [DirichletBC(V, Constant((0.,0.,0.)),left),
DirichletBC(V, Constant((0.,0.,0.)),right),
DirichletBC(V, Constant((0.,0.,0.)),top),
DirichletBC(V, Constant((0.,0.,0.)),bottom)]

# Matrices
k_form = inner(sigma(du),eps(u_))*dx
l_form = Constant(1.)*u_[0]*dx
K = PETScMatrix()
b = PETScVector()
assemble_system(k_form,l_form,bc,A_tensor=K,b_tensor=b)

m_form = rho*dot(du,u_)*dx
M = PETScMatrix()
assemble(m_form, tensor=M)

# Eigenvalues/Eigensolver
eigensolver = SLEPcEigenSolver(K,M)
eigensolver.parameters['problem_type'] = 'gen_hermitian'
```

```
    #eigensolver.parameters['spectrum'] = 'smallest real'
    eigensolver.parameters['spectral_transform'] = 'shift-and-invert'
    eigensolver.parameters['spectral_shift'] = 0.
    N_eig = 2
    eigensolver.solve(N_eig)
    #print (eigensolver.parameters.str(True))

    # Export results
    file_results = XDMFFile('acc_modal_analysis.xdmf')
    file_results.parameters['flush_output'] = True
    file_results.parameters['functions_share_mesh'] = True

    r1,c1,rx1,cx1 = eigensolver.get_eigenpair(0)
    u = Function(V)
    u.vector()[:] = rx1
    file_results.write(u,0)

    # Extraction
    for i in range(N_eig):
        r,c,rx,cx = eigensolver.get_eigenpair(i)
        freq = sqrt(r)/2/pi
        print('Mode:',i,'   ','Freq:',freq,'[Hz]')


    freq_final = sqrt(r1)/2/pi
    return freq_final
```

Listing I.4: Electronic domain simulation script

```
# MEMS accelerometer electrical domain simulation
# @ruiesteves

# Imports
import acc_disp



# Constants
e0 = 8.85e-12 # Permitivity of free space
er = 1 # Relative permitivity of dielectric, in this case air
small_gap = 22*scale # Distance between electrodes and proof mass


# Functions

def main(suspension_beam_width,proof_mass_length):
```

```python
    disp = acc_disp.disp(suspension_beam_width,proof_mass_length)
    A = (proof_mass_length)**2
    def top_capacitance():
        C = (e0*er*A)/(small_gap + disp)
        return C

    def bot_capacitance():
        C = (e0*er*A)/(small_gap - disp)
        return C

    def capacitance_total():
        c_total = bot_capacitance() - top_capacitance()
        print(c_total*1e15,"fF")
        return c_total

    def c2v():
        v = 2 * capacitance_total() * 2.5 * (1/300e-15)
        print("Output voltage:",v*1e3,"mV")
        return v

    voltage = c2v()
    return voltage
```

Listing I.5: Damping calculation script

```python
# MEMS accelerometer squeeze film damping calculation script
# @ruiesteves

import math

epsilon0 = 8.85e-12
scale = 1e-6
mu = 1.86e-5 # the mean viscosity of the medium
lamb = 0.067e-6 # mean free path
small_gap = 22*scale
thickness = 320*scale
rho = 2329

def q_factor(suspension_beam_width,proof_mass_length,sense_frequency):
    A = proof_mass_length**2
    mass_sense = A * thickness * rho
    Kn = lamb/small_gap
    mu_eff = mu/(1+9.638*Kn**1.159)
    Pa = 101.3e3
    c = 1
    squeeze_number = (12*mu_eff*2*math.pi*L_sense**2)/(Pa*small_gap**2)
```

```
    sum = 0
    for m in range(1,10,2):
        for n in (1,10,2):
            sum = sum + (m**2 + c**2 * n**2)/((m*n)**2 * ((m**2 + c**2 * n**2)**2
                ↪ + (squeeze_number**2 / math.pi**4)))

    F_damping = ((64*squeeze_number*Pa*A)/(math.pi**6 * small_gap)) * sum
    c_sense = F_damping
    q_factor = mass_sense * sense_frequency*2*math.pi / c_sense

    return q_factor
```

Listing I.6: Genetic algorithm script

```
# Python Accelerometer GA
# @ruiesteves

# Imports
import acc_geo
import acc_disp
import acc_elec
import acc_modal
import acc_damping
import numpy as np
import math as math
import random as rand
import copy

# Initial parameters of the device to be optimized
scale = 1e-6
suspension_beam_width = 350*scale
proof_mass_length = 2400*scale

initial = [suspension_beam_width,proof_mass_length]

# Classes
class GA_device:

    def __init__(self,id):
        self.list_parameters = []
        self.id = id

    def calc_sensitivity(self):
        list = self.list_parameters
        self.sensitivity = acc_elec.main(list[0],list[1])

```

```
    def calc_freq(self):
        list = self.list_parameters
        self.freq = acc_modal.main(list[0],list[1])

    def calc_qfactor(self):
        list = self.list_parameters
        self.qfactor = acc_damping.q_factor(list[0],list[1],self.freq)

    def calc_fom(self):
        list = self.list_parameters
        try:
            acc_geo.build(list[0],list[1])
            self.calc_sensitivity()
            self.calc_freq()
            self.calc_qfactor()
            self.fom = self.freq * self.sensitivity * self.qfactor * (1/1000)
        except:
            print("Geometry became invalid for device",self.id)
            self.fom = 0


class GA: # GA class, initiated with a list of devices, a list of mutation chances and a list of mutation relative size

    def __init__(self,list_devices,mutation_chance,mutation_size):
        self.list_devices = list_devices # Must be a list of GA_devices
        self.mutation_chance = mutation_chance # A list, with different (or not) mutation chances for each parameter
        self.mutation_size = mutation_size # Same as above, this time for mutation_sizes (IMPORTANT to check)

    def mutate(self,dev): # The mutation function, mutating the parameters according to their mutation chance and size
        le = len(dev.list_parameters)
        for i in range(le):
            if rand.uniform(0,1) < self.mutation_chance[i]:
                if rand.uniform(0,1) < 0.5:
                    dev.list_parameters[i] = dev.list_parameters[i] + dev.list_parameters[i]*self.mutation_size[i]
                else:
                    dev.list_parameters[i] = dev.list_parameters[i] - dev.list_parameters[i]*self.mutation_size[i]

    def reproduce(self,top_25):
        new_population = []
```

```

        for dev in top_25: # Passing the best 25 devices to the next generation
            new_population.append(dev)

        for dev in top_25: # Copying and mutating the best 25 devices to the next generation
            new_dev = copy.deepcopy(dev)
            self.mutate(new_dev)
            new_population.append(new_dev)

        for i in range(len(self.list_devices)//2): # Randomly mutating and passing half of the population to the next generation
            new_dev_r = copy.deepcopy(self.list_devices[i])
            self.mutate(new_dev_r)
            new_population.append(new_dev_r)

        return new_population


    def one_generation(self):

        for dev in self.list_devices:
            dev.calc_fom()

        le = len(self.list_devices)
        scores = [self.list_devices[i].fom for i in range(le)]
        max = np.amax(scores)
        print(scores)
        print(max)

        top_25_index = list(np.argsort(scores))[3*(le//4):le]
        top_25 = [self.list_devices[i] for i in top_25_index][::-1]

        self.list_devices = self.reproduce(top_25)





# Script
print("Genetic algorithm optimization for MEMS accelerometer")
num_pop = int(input("Size of the population: "))
num_gen = int(input("Number of generations: "))

initial_pop = []
```

```
for i in range(num_pop):
    initial_pop.append(GA_device(i))
    for par in range(len(initial)):
        initial_pop[i].list_parameters.append(initial[par])
    initial_pop[i].calc_fom()

init_ga = GA(initial_pop,[0.6,0.6],[0.13,0.0143])

for i in range(num_gen):
    init_ga.one_generation()
    for dev in init_ga.list_devices:
        print("\n","For Device number",dev.id,":")
        print(dev.sensitivity*1e3,"mV/g")
        print(dev.freq,"Hz")
        print(dev.qfactor,"Q-factor")
        print(dev.fom,"FOM")

max_score = 0


for dev in init_ga.list_devices:
    if dev.fom > max_score:
        max_score = dev.fom

print("Maximum FOM:",max_score)
```

Annex II

Software implementation on MEMS gyroscope

In the following listings, the program implementation for a MEMS gyroscope is displayed. The code makes it possible to obtain the displacement due to an actuation force or coriolis force via FEM - however, it is recommended to make use of the displacement equations for long optimization runs.

Listing II.1: Geometry building block

```
# MEMS Gyroscope 3D Geometry
# @ruiesteves

# Imports
import pygmsh as pg # geometry & meshing definition
import meshio # meshing export

# Helper functions
def mirror_quarter_helper(mirror_quarter,x_point):
    dist = mirror_quarter - x_point
    x_new = mirror_quarter + dist
    return x_new

def mirror_half_helper(drive_frame_beam_h,y_point):
    dist = drive_frame_beam_h - y_point
    y_new = drive_frame_beam_h + dist
    return y_new

# Functions
def build(serpentine_width,proof_mass_beam_w,proof_mass_beam_h,
proof_mass_w,proof_mass_h,sense_comb_finger_h):

    # Constants
    scale = 1e-6
    cl = 80*scale
    small_cl = 9*scale
    thickness = 50*scale
    small_gap = 3*scale
```

```
large_gap = 4*small_gap
drive_anchor_width = 124*scale
drive_anchor_height = 126*scale
drive_serpentine_connector_w = 21*scale
drive_serpentine_connector_h = 24*scale
drive_serpentine_beam_h = 194*scale
drive_serpentine_connector2_w = 17*scale
drive_serpentine_connector2_h = 21*scale
drive_serpentine_beam2_x = drive_anchor_width + drive_serpentine_connector_w + serpentine_width + drive_serpentine_connector2_w
drive_serpentine_connector3_w = 17*scale
drive_serpentine_connector3_h = 24*scale
drive_frame_beam_w = 56*scale
drive_frame_beam_h = 485*scale
drive_frame_connector_w = 72*scale
drive_frame_connector_h = 73*scale
drive_frame_serpent_beam_w = 171*scale
drive_frame_serpent_connector_x = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + drive_frame_serpent_beam_w - drive_serpentine_connector2_h
drive_frame_base_w = 309*scale
drive_frame_base_h = 41*scale
sense_comb_finger_dist = (small_gap + large_gap + sense_comb_finger_h)
proof_mass_fingers_pole_w = 15*scale
proof_mass_fingers_pole_h = (3*(sense_comb_finger_dist+sense_comb_finger_h))
sense_comb_finger_w = 243*scale
sense_comb_finger_num = 3
drive_comb_finger_w = 48*scale
drive_comb_finger_h = 9*scale
drive_comb_finger_dist = (small_gap + large_gap + drive_comb_finger_h)
drive_comb_finger_num = 8
mirror_quarter = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w
mirror_gyro = mirror_quarter*2 - drive_anchor_width/2
drive_coupling_beam_w = 118.5*scale
drive_coupling_beam_h = 21*scale
drive_coupling_dist = 42.75*scale
drive_coupling_beam_vert_w = 9*scale
drive_coupling_dist2 = 95*scale
drive_coupling_beam_vert_h = drive_coupling_dist2*2 + drive_coupling_beam_h
drive_coupling_connector_w = 15*scale
drive_coupling_connector_h = 12*scale

```

```
# Geometry build
geom = pg.opencascade.Geometry()

# Drive anchor
p1 = [0,0,0]
p2 = [drive_anchor_width,drive_anchor_height,thickness]
drive_anchor = geom.add_box(p1,p2,char_length=cl)

# Drive serpentine connector
p3 = [drive_anchor_width,0,0]
p4 = [drive_serpentine_connector_w,drive_serpentine_connector_h,thickness]
drive_serpentine_connector = geom.add_box(p3,p4,char_length=cl)

# Drive serpentine beam
p5 = [drive_anchor_width + drive_serpentine_connector_w,0,0]
p6 = [serpentine_width,drive_serpentine_beam_h,thickness]
drive_serpentine_beam = geom.add_box(p5,p6,char_length=cl)

# Drive serpentine connector 2
p7 = [drive_anchor_width + drive_serpentine_connector_w + serpentine_width, drive_serpentine_beam_h - drive_serpentine_connector2_h,0]
p8 = [drive_serpentine_connector2_w,drive_serpentine_connector2_h,thickness]
drive_serpentine_connector2 = geom.add_box(p7,p8,char_length=cl)

# Drive serpentine beam 2
p9 = [drive_serpentine_beam2_x,0,0]
p10 = [serpentine_width,drive_serpentine_beam_h,thickness]
drive_serpentine_beam2 = geom.add_box(p9,p10,char_length=cl)

# Drive serpentine connector 3
p11 = [drive_serpentine_beam2_x + serpentine_width,0,0]
p12 = [drive_serpentine_connector3_w,drive_serpentine_connector3_h,thickness]
drive_serpentine_connector3 = geom.add_box(p11,p12,char_length=cl)

# Drive frame beam
p13 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w,0,0]
p14 = [drive_frame_beam_w,drive_frame_beam_h,thickness]
drive_frame_beam = geom.add_box(p13,p14,char_length=cl)

# Drive frame connector
p15 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w,0,0]
p16 = [drive_frame_connector_w,drive_frame_connector_h,thickness]
```

```
drive_frame_connector = geom.add_box(p15,p16,char_length=cl)

# Drive frame serpent beam
p17 = [drive_serpentine_beam2_x + serpentine_width +
    ↪ drive_serpentine_connector3_w+drive_frame_beam_w +
    ↪ drive_frame_connector_w,drive_frame_connector_h - serpentine_width,0]
p18 = [drive_frame_serpent_beam_w,serpentine_width,thickness]
drive_frame_serpent_beam = geom.add_box(p17,p18,char_length=cl)

# Drive frame serpent connector
p19 = [drive_frame_serpent_connector_x,drive_frame_connector_h,0]
p20 = [drive_serpentine_connector2_h,drive_serpentine_connector2_w,thickness
    ↪ ]
drive_frame_serpent_connector = geom.add_box(p19,p20,char_length=cl)

# Drive frame serpent beam 2
p21 = [drive_serpentine_beam2_x + serpentine_width +
    ↪ drive_serpentine_connector3_w+drive_frame_beam_w +
    ↪ drive_frame_connector_w,drive_frame_connector_h +
    ↪ drive_serpentine_connector2_w,0]
p22 = [drive_frame_serpent_beam_w,serpentine_width,thickness]
drive_frame_serpent_beam2 = geom.add_box(p21,p22,char_length=cl)

# Drive frame base
p23 = [drive_serpentine_beam2_x + serpentine_width +
    ↪ drive_serpentine_connector3_w+drive_frame_beam_w +
    ↪ drive_frame_connector_w,0,0]
p24 = [drive_frame_base_w,drive_frame_base_h,thickness]
drive_frame_base = geom.add_box(p23,p24,char_length=cl)

# Proof mass beam
p25 = [drive_serpentine_beam2_x + serpentine_width +
    ↪ drive_serpentine_connector3_w+drive_frame_beam_w +
    ↪ drive_frame_connector_w - proof_mass_beam_w,drive_frame_connector_h +
    ↪ drive_serpentine_connector2_w,0]
p26 = [proof_mass_beam_w,proof_mass_beam_h,thickness]
proof_mass_beam = geom.add_box(p25,p26,char_length=cl)

# Proof mass
p27 = [drive_serpentine_beam2_x + serpentine_width +
    ↪ drive_serpentine_connector3_w+drive_frame_beam_w +
    ↪ drive_frame_connector_w, drive_frame_connector_h +
    ↪ drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h,0]
p28 = [proof_mass_w,proof_mass_h,thickness]
proof_mass = geom.add_box(p27,p28,char_length=cl)
```

```
# Proof mass fingers pole
p29 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w,
drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h,0]
p30 = [proof_mass_fingers_pole_w,proof_mass_fingers_pole_h,thickness]
proof_mass_fingers_pole = geom.add_box(p29,p30,char_length=cl)

# Sense comb finger array
sense_comb_finger_array = []
for i in range(sense_comb_finger_num):
    p31 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w - sense_comb_finger_w,
    drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h + (i)*(sense_comb_finger_h + sense_comb_finger_dist),0]
    p32 = [sense_comb_finger_w,sense_comb_finger_h,thickness]
    name = "".join(["sense_comb_finger",str(i)])
    name = geom.add_box(p31,p32,char_length=cl)
    sense_comb_finger_array.append(name)
sense_comb_finger_complete = geom.boolean_union(sense_comb_finger_array)

# Drive comb finger array
drive_comb_finger_array = []
for i in range(drive_comb_finger_num):
    p33 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w - drive_comb_finger_w,
    drive_frame_beam_h - drive_comb_finger_dist/2 - drive_comb_finger_h - i*(drive_comb_finger_h + drive_comb_finger_dist),0]
    p34 = [drive_comb_finger_w,drive_comb_finger_h,thickness]
    name = "".join(["drive_comb_finger",str(i)])
    name = geom.add_box(p33,p34,char_length=cl)
    drive_comb_finger_array.append(name)
drive_comb_finger_complete = geom.boolean_union(drive_comb_finger_array)


# Quarter union
print("1st union starts here")
quarter = geom.boolean_union([drive_anchor,drive_serpentine_connector, drive_serpentine_beam,
```

```
    drive_serpentine_connector2,drive_serpentine_beam2, drive_serpentine_connector3,drive_frame_beam,
    drive_frame_connector,drive_frame_serpent_beam,drive_frame_serpent_connector, drive_frame_serpent_beam2,
    drive_frame_base,proof_mass_beam,proof_mass,proof_mass_fingers_pole, sense_comb_finger_complete,
    drive_comb_finger_complete])


    # QUARTER MIRROR STARTS HERE
    # Drive anchor
    p35 = [mirror_quarter_helper(mirror_quarter,0),0,0]
    print("ANCHOR:",p35)
    p36 = [-drive_anchor_width,drive_anchor_height,thickness]
    drive_anchorq = geom.add_box(p35,p36,char_length=cl)

    # Drive serpentine connector
    p37 = [mirror_quarter_helper(mirror_quarter,drive_anchor_width),0,0]
    p38 = [-drive_serpentine_connector_w,drive_serpentine_connector_h,thickness]
    drive_serpentine_connectorq = geom.add_box(p37,p38,char_length=cl)

    # Drive serpentine beam
    p39 = [mirror_quarter_helper(mirror_quarter,drive_anchor_width + drive_serpentine_connector_w),0,0]
    p40 = [-serpentine_width,drive_serpentine_beam_h,thickness]
    drive_serpentine_beamq = geom.add_box(p39,p40,char_length=cl)

    # Drive serpentine connector 2
    p41 = [mirror_quarter_helper(mirror_quarter,drive_anchor_width + drive_serpentine_connector_w + serpentine_width), drive_serpentine_beam_h - drive_serpentine_connector2_h,0]
    p42 = [-drive_serpentine_connector2_w,drive_serpentine_connector2_h, thickness]
    drive_serpentine_connector2q = geom.add_box(p41,p42,char_length=cl)

    # Drive serpentine beam 2
    p9 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x),0,0]
    p10 = [-serpentine_width,drive_serpentine_beam_h,thickness]
    drive_serpentine_beam2q = geom.add_box(p9,p10,char_length=cl)

    # Drive serpentine connector 3
    p11 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width),0,0]
    p12 = [-drive_serpentine_connector3_w,drive_serpentine_connector3_h, thickness]
```

```python
    drive_serpentine_connector3q = geom.add_box(p11,p12,char_length=cl)

    # Drive frame beam
    p13 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w),0,0]
    p14 = [-drive_frame_beam_w,drive_frame_beam_h,thickness]
    drive_frame_beamq = geom.add_box(p13,p14,char_length=cl)

    # Drive frame connector
    p15 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w),0,0]
    p16 = [-drive_frame_connector_w,drive_frame_connector_h,thickness]
    drive_frame_connectorq = geom.add_box(p15,p16,char_length=cl)

    # Drive frame serpent beam
    p17 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w),drive_frame_connector_h - serpentine_width,0]
    p18 = [-drive_frame_serpent_beam_w,serpentine_width,thickness]
    drive_frame_serpent_beamq = geom.add_box(p17,p18,char_length=cl)

    # Drive frame serpent connector
    p19 = [mirror_quarter_helper(mirror_quarter,drive_frame_serpent_connector_x),drive_frame_connector_h,0]
    p20 = [-drive_serpentine_connector2_h,drive_serpentine_connector2_w,thickness]
    drive_frame_serpent_connectorq = geom.add_box(p19,p20,char_length=cl)

    # Drive frame serpent beam 2
    p21 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w),drive_frame_connector_h + drive_serpentine_connector2_w,0]
    p22 = [-drive_frame_serpent_beam_w,serpentine_width,thickness]
    drive_frame_serpent_beam2q = geom.add_box(p21,p22,char_length=cl)

    # Drive frame base
    p23 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w),0,0]
    p24 = [-drive_frame_base_w,drive_frame_base_h,thickness]
    drive_frame_baseq = geom.add_box(p23,p24,char_length=cl)

    # Proof mass beam
```

```
p25 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w - proof_mass_beam_w),drive_frame_connector_h + drive_serpentine_connector2_w,0]
p26 = [-proof_mass_beam_w,proof_mass_beam_h,thickness]
proof_mass_beamq = geom.add_box(p25,p26,char_length=cl)

# Proof mass
p27 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w), drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h,0]
p28 = [-proof_mass_w,proof_mass_h,thickness]
proof_massq = geom.add_box(p27,p28,char_length=cl)

# Proof mass fingers pole
p29 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w),
drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h,0]
p30 = [-proof_mass_fingers_pole_w,proof_mass_fingers_pole_h,thickness]
proof_mass_fingers_poleq = geom.add_box(p29,p30,char_length=cl)

# Sense comb finger array
sense_comb_finger_array = []
for i in range(sense_comb_finger_num):
    p31 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+ drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w - sense_comb_finger_w),
    drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h + (i) *(sense_comb_finger_h + sense_comb_finger_dist),0]
    p32 = [-sense_comb_finger_w,sense_comb_finger_h,thickness]
    name = "".join(["sense_comb_finger",str(i)])
    name = geom.add_box(p31,p32,char_length=cl)
    sense_comb_finger_array.append(name)
sense_comb_finger_completeq = geom.boolean_union(sense_comb_finger_array)

# Drive comb finger array
drive_comb_finger_array = []
for i in range(drive_comb_finger_num):
```

```python
        p33 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w - drive_comb_finger_w),
        drive_frame_beam_h - drive_comb_finger_dist/2 - drive_comb_finger_h - i*(drive_comb_finger_h + drive_comb_finger_dist),0]
        p34 = [-drive_comb_finger_w,drive_comb_finger_h,thickness]
        name = "".join(["drive_comb_finger",str(i)])
        name = geom.add_box(p33,p34,char_length=cl)
        drive_comb_finger_array.append(name)
    drive_comb_finger_completeq = geom.boolean_union(drive_comb_finger_array)

    print("2nd union starts here")
    # Quarter union
    quarter_right = geom.boolean_union([drive_anchorq, drive_serpentine_connectorq,drive_serpentine_beamq,
    drive_serpentine_connector2q,drive_serpentine_beam2q, drive_serpentine_connector3q,drive_frame_beamq,
    drive_frame_connectorq,drive_frame_serpent_beamq, drive_frame_serpent_connectorq,drive_frame_serpent_beam2q,
    drive_frame_baseq,proof_mass_beamq,proof_massq,proof_mass_fingers_poleq, sense_comb_finger_completeq,
    drive_comb_finger_completeq])



    # HALF MIRRORING STARTS FROM HERE
    # Drive anchor
    p1 = [0,mirror_half_helper(drive_frame_beam_h,0),0]
    p2 = [drive_anchor_width,-drive_anchor_height,thickness]
    drive_anchorh = geom.add_box(p1,p2,char_length=cl)

    # Drive serpentine connector
    p3 = [drive_anchor_width,mirror_half_helper(drive_frame_beam_h,0),0]
    p4 = [drive_serpentine_connector_w,-drive_serpentine_connector_h,thickness]
    drive_serpentine_connectorh = geom.add_box(p3,p4,char_length=cl)

    # Drive serpentine beam
    p5 = [drive_anchor_width + drive_serpentine_connector_w,mirror_half_helper(drive_frame_beam_h,0),0]
    p6 = [serpentine_width,-drive_serpentine_beam_h,thickness]
    drive_serpentine_beamh = geom.add_box(p5,p6,char_length=cl)

    # Drive serpentine connector 2
```

```
    p7 = [drive_anchor_width + drive_serpentine_connector_w + serpentine_width, mirror_half_helper(drive_frame_beam_h,drive_serpentine_beam_h - drive_serpentine_connector2_h),0]
    p8 = [drive_serpentine_connector2_w,-drive_serpentine_connector2_h,thickness]
    drive_serpentine_connector2h = geom.add_box(p7,p8,char_length=cl)

    # Drive serpentine beam 2
    p9 = [drive_serpentine_beam2_x,mirror_half_helper(drive_frame_beam_h,0),0]
    p10 = [serpentine_width,-drive_serpentine_beam_h,thickness]
    drive_serpentine_beam2h = geom.add_box(p9,p10,char_length=cl)

    # Drive serpentine connector 3
    p11 = [drive_serpentine_beam2_x + serpentine_width,mirror_half_helper(drive_frame_beam_h,0),0]
    p12 = [drive_serpentine_connector3_w,-drive_serpentine_connector3_h,thickness]
    drive_serpentine_connector3h = geom.add_box(p11,p12,char_length=cl)

    # Drive frame beam
    p13 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w,mirror_half_helper(drive_frame_beam_h,0),0]
    p14 = [drive_frame_beam_w,-drive_frame_beam_h,thickness]
    drive_frame_beamh = geom.add_box(p13,p14,char_length=cl)

    # Drive frame connector
    p15 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w,mirror_half_helper(drive_frame_beam_h,0),0]
    p16 = [drive_frame_connector_w,-drive_frame_connector_h,thickness]
    drive_frame_connectorh = geom.add_box(p15,p16,char_length=cl)

    # Drive frame serpent beam
    p17 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w,mirror_half_helper(drive_frame_beam_h, drive_frame_connector_h - serpentine_width),0]
    p18 = [drive_frame_serpent_beam_w,-serpentine_width,thickness]
    drive_frame_serpent_beamh = geom.add_box(p17,p18,char_length=cl)

    # Drive frame serpent connector
    p19 = [drive_frame_serpent_connector_x,mirror_half_helper(drive_frame_beam_h, drive_frame_connector_h),0]
```

```
p20 = [drive_serpentine_connector2_h,-drive_serpentine_connector2_w, thickness]
drive_frame_serpent_connectorh = geom.add_box(p19,p20,char_length=cl)

# Drive frame serpent beam 2
p21 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w,mirror_half_helper(drive_frame_beam_h, drive_frame_connector_h + drive_serpentine_connector2_w),0]
p22 = [drive_frame_serpent_beam_w,-serpentine_width,thickness]
drive_frame_serpent_beam2h = geom.add_box(p21,p22,char_length=cl)

# Drive frame base
p23 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w,mirror_half_helper(drive_frame_beam_h,0),0]
p24 = [drive_frame_base_w,-drive_frame_base_h,thickness]
drive_frame_baseh = geom.add_box(p23,p24,char_length=cl)

# Proof mass beam
p25 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w - proof_mass_beam_w,mirror_half_helper( drive_frame_beam_h,drive_frame_connector_h + drive_serpentine_connector2_w),0]
p26 = [proof_mass_beam_w,-proof_mass_beam_h,thickness]
proof_mass_beamh = geom.add_box(p25,p26,char_length=cl)

# Proof mass
p27 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w, mirror_half_helper(drive_frame_beam_h, drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h),0]
p28 = [proof_mass_w,-proof_mass_h,thickness]
proof_massh = geom.add_box(p27,p28,char_length=cl)

# Proof mass fingers pole
p29 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w,
mirror_half_helper(drive_frame_beam_h,drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h),0]
p30 = [proof_mass_fingers_pole_w,-proof_mass_fingers_pole_h,thickness]
```

```
    proof_mass_fingers_poleh = geom.add_box(p29,p30,char_length=cl)

    # Sense comb finger array
    sense_comb_finger_array = []
    for i in range(sense_comb_finger_num):
        p31 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w - sense_comb_finger_w,
        mirror_half_helper(drive_frame_beam_h,drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h + (i)*(sense_comb_finger_h + sense_comb_finger_dist)),0]
        p32 = [sense_comb_finger_w,-sense_comb_finger_h,thickness]
        name = "".join(["sense_comb_finger",str(i)])
        name = geom.add_box(p31,p32,char_length=cl)
        sense_comb_finger_array.append(name)
    sense_comb_finger_completeh = geom.boolean_union(sense_comb_finger_array)

    # Drive comb finger array
    drive_comb_finger_array = []
    for i in range(drive_comb_finger_num):
        p33 = [drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w - drive_comb_finger_w,
        mirror_half_helper(drive_frame_beam_h,drive_frame_beam_h - drive_comb_finger_dist/2 - drive_comb_finger_h - i*(drive_comb_finger_h + drive_comb_finger_dist)),0]
        p34 = [drive_comb_finger_w,-drive_comb_finger_h,thickness]
        name = "".join(["drive_comb_finger",str(i)])
        name = geom.add_box(p33,p34,char_length=cl)
        drive_comb_finger_array.append(name)
    drive_comb_finger_completeh = geom.boolean_union(drive_comb_finger_array)


    print("3rd union starts here")
    # Quarter union
    quarter_h = geom.boolean_union([drive_anchorh,drive_serpentine_connectorh, drive_serpentine_beamh,
    drive_serpentine_connector2h,drive_serpentine_beam2h, drive_serpentine_connector3h,drive_frame_beamh,
    drive_frame_connectorh,drive_frame_serpent_beamh, drive_frame_serpent_connectorh,drive_frame_serpent_beam2h,
    drive_frame_baseh,proof_mass_beamh,proof_massh,proof_mass_fingers_poleh, sense_comb_finger_completeh,
    drive_comb_finger_completeh])
```

```
# QUARTER MIRROR STARTS HERE
# Drive anchor
p35 = [mirror_quarter_helper(mirror_quarter,0),mirror_half_helper(drive_frame_beam_h,0),0]
p36 = [-drive_anchor_width,-drive_anchor_height,thickness]
drive_anchorqh = geom.add_box(p35,p36,char_length=cl)

# Drive serpentine connector
p37 = [mirror_quarter_helper(mirror_quarter,drive_anchor_width),mirror_half_helper(drive_frame_beam_h,0),0]
p38 = [-drive_serpentine_connector_w,-drive_serpentine_connector_h,thickness]
drive_serpentine_connectorqh = geom.add_box(p37,p38,char_length=cl)

# Drive serpentine beam
p39 = [mirror_quarter_helper(mirror_quarter,drive_anchor_width + drive_serpentine_connector_w),mirror_half_helper(drive_frame_beam_h,0),0]
p40 = [-serpentine_width,-drive_serpentine_beam_h,thickness]
drive_serpentine_beamqh = geom.add_box(p39,p40,char_length=cl)

# Drive serpentine connector 2
p41 = [mirror_quarter_helper(mirror_quarter,drive_anchor_width + drive_serpentine_connector_w + serpentine_width),mirror_half_helper(drive_frame_beam_h,drive_serpentine_beam_h - drive_serpentine_connector2_h),0]
p42 = [-drive_serpentine_connector2_w,-drive_serpentine_connector2_h,thickness]
drive_serpentine_connector2qh = geom.add_box(p41,p42,char_length=cl)

# Drive serpentine beam 2
p9 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x),mirror_half_helper(drive_frame_beam_h,0),0]
p10 = [-serpentine_width,-drive_serpentine_beam_h,thickness]
drive_serpentine_beam2qh = geom.add_box(p9,p10,char_length=cl)

# Drive serpentine connector 3
p11 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width),mirror_half_helper(drive_frame_beam_h,0),0]
p12 = [-drive_serpentine_connector3_w,-drive_serpentine_connector3_h,thickness]
drive_serpentine_connector3qh = geom.add_box(p11,p12,char_length=cl)
```

```
    # Drive frame beam
    p13 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w),mirror_half_helper(drive_frame_beam_h,0),0]
    p14 = [-drive_frame_beam_w,-drive_frame_beam_h,thickness]
    drive_frame_beamqh = geom.add_box(p13,p14,char_length=cl)

    # Drive frame connector
    p15 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w), mirror_half_helper(drive_frame_beam_h,0),0]
    p16 = [-drive_frame_connector_w,-drive_frame_connector_h,thickness]
    drive_frame_connectorqh = geom.add_box(p15,p16,char_length=cl)

    # Drive frame serpent beam
    p17 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w),mirror_half_helper(drive_frame_beam_h, drive_frame_connector_h - serpentine_width),0]
    p18 = [-drive_frame_serpent_beam_w,-serpentine_width,thickness]
    drive_frame_serpent_beamqh = geom.add_box(p17,p18,char_length=cl)

    # Drive frame serpent connector
    p19 = [mirror_quarter_helper(mirror_quarter,drive_frame_serpent_connector_x), mirror_half_helper(drive_frame_beam_h,drive_frame_connector_h),0]
    p20 = [-drive_serpentine_connector2_h,-drive_serpentine_connector2_w, thickness]
    drive_frame_serpent_connectorqh = geom.add_box(p19,p20,char_length=cl)

    # Drive frame serpent beam 2
    p21 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w),mirror_half_helper(drive_frame_beam_h, drive_frame_connector_h + drive_serpentine_connector2_w),0]
    p22 = [-drive_frame_serpent_beam_w,-serpentine_width,thickness]
    drive_frame_serpent_beam2qh = geom.add_box(p21,p22,char_length=cl)

    # Drive frame base
    p23 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w),mirror_half_helper(drive_frame_beam_h,0),0]
    p24 = [-drive_frame_base_w,-drive_frame_base_h,thickness]
    drive_frame_baseqh = geom.add_box(p23,p24,char_length=cl)

    # Proof mass beam
```

```
p25 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w - proof_mass_beam_w),mirror_half_helper( drive_frame_connector_h + drive_serpentine_connector2_w),0]
p26 = [-proof_mass_beam_w,-proof_mass_beam_h,thickness]
proof_mass_beamqh = geom.add_box(p25,p26,char_length=cl)

# Proof mass
p27 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w), mirror_half_helper(drive_frame_beam_h, drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h),0]
p28 = [-proof_mass_w,-proof_mass_h,thickness]
proof_massqh = geom.add_box(p27,p28,char_length=cl)

# Proof mass fingers pole
p29 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w),
mirror_half_helper(drive_frame_beam_h,drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h),0]
p30 = [-proof_mass_fingers_pole_w,-proof_mass_fingers_pole_h,thickness]
proof_mass_fingers_poleqh = geom.add_box(p29,p30,char_length=cl)

# Sense comb finger array
sense_comb_finger_array = []
for i in range(sense_comb_finger_num):
    p31 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+ drive_frame_beam_w + drive_frame_connector_w + proof_mass_w - proof_mass_fingers_pole_w - sense_comb_finger_w),
    mirror_half_helper(drive_frame_beam_h,drive_frame_connector_h + drive_serpentine_connector2_w + proof_mass_beam_h - proof_mass_h - proof_mass_fingers_pole_h + (i)*(sense_comb_finger_h + sense_comb_finger_dist)),0]
    p32 = [-sense_comb_finger_w,-sense_comb_finger_h,thickness]
    name = "".join(["sense_comb_finger",str(i)])
    name = geom.add_box(p31,p32,char_length=cl)
    sense_comb_finger_array.append(name)
sense_comb_finger_completeqh = geom.boolean_union(sense_comb_finger_array)

# Drive comb finger array
drive_comb_finger_array = []
```

```
    for i in range(drive_comb_finger_num):
        p33 = [mirror_quarter_helper(mirror_quarter,drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w - drive_comb_finger_w),
        mirror_half_helper(drive_frame_beam_h,drive_frame_beam_h - drive_comb_finger_dist/2 - drive_comb_finger_h - i*(drive_comb_finger_h + drive_comb_finger_dist)),0]
        p34 = [-drive_comb_finger_w,-drive_comb_finger_h,thickness]
        name = "".join(["drive_comb_finger",str(i)])
        name = geom.add_box(p33,p34,char_length=cl)
        drive_comb_finger_array.append(name)
    drive_comb_finger_completeqh = geom.boolean_union(drive_comb_finger_array)

    print("4th union starts here")
    # Quarter union
    quarter_qh = geom.boolean_union([drive_anchorqh,drive_serpentine_connectorqh, drive_serpentine_beamqh,
    drive_serpentine_connector2qh,drive_serpentine_beam2qh, drive_serpentine_connector3qh,drive_frame_beamqh,
    drive_frame_connectorqh,drive_frame_serpent_beamqh, drive_frame_serpent_connectorqh,drive_frame_serpent_beam2qh,
    drive_frame_baseqh,proof_mass_beamqh,proof_massqh,proof_mass_fingers_poleqh, sense_comb_finger_completeqh,
    drive_comb_finger_completeqh])





    print("Final union starts here")
    # Complete union
    complete = geom.boolean_union([quarter,quarter_right,quarter_h,quarter_qh])





    mesh = pg.generate_mesh(geom,gmsh_path="/home/ruiesteves/Documents/Tese/MechanicalModel/gmsh-4.5.2-Linux64/bin/gmsh")
    meshio.write("gyroscope.xml",mesh)
    #meshio.write("antiphase_geo.mesh",mesh)

    return mesh
```

Listing II.2: Displacement simulation script

```python
# MEMS Gyroscope displacement simulation script
# @ruiesteves

# Imports
from __future__ import print_function
from dolfin import *
import math
import gyro_elec
import gyro_damping

# Helper functions
def mirror_quarter_helper(mirror_quarter,x_point):
    dist = mirror_quarter - x_point
    x_new = mirror_quarter + dist
    return x_new

def mirror_half_helper(drive_frame_beam_h,y_point):
    dist = drive_frame_beam_h - y_point
    y_new = drive_frame_beam_h + dist
    return y_new


# Constants
E = Constant(170e9)
nu = Constant(0.28)
rho = 2329
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)


# The user can choose between the two ways of calculating displacement: FEM simulation or equations. For long optimization runs, the equations approach is preferred.
def disp_equations(serpentine_width,proof_mass_beam_w,proof_mass_beam_h,proof_mass_w,proof_mass_h,sense_comb_finger_h,q_factor_sense,q_factor_drive,drive_frequency,sense_frequency):

    # Constants
    scale = 1e-6
    cl = 80*scale
    small_cl = 9*scale
    thickness = 50*scale
    small_gap = 3*scale
    large_gap = 4*small_gap
    drive_anchor_width = 124*scale
```

```
drive_anchor_height = 126*scale
drive_serpentine_connector_w = 21*scale
drive_serpentine_connector_h = 24*scale
drive_serpentine_beam_h = 194*scale
drive_serpentine_connector2_w = 17*scale
drive_serpentine_connector2_h = 21*scale
drive_serpentine_beam2_x = drive_anchor_width + drive_serpentine_connector_w + serpentine_width + drive_serpentine_connector2_w
drive_serpentine_connector3_w = 17*scale
drive_serpentine_connector3_h = 24*scale
drive_frame_beam_w = 56*scale
drive_frame_beam_h = 485*scale
drive_frame_connector_w = 72*scale
drive_frame_connector_h = 73*scale
drive_frame_serpent_beam_w = 171*scale
drive_frame_serpent_connector_x = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + drive_frame_serpent_beam_w - drive_serpentine_connector2_h
drive_frame_base_w = 309*scale
drive_frame_base_h = 41*scale
sense_comb_finger_dist = (small_gap + large_gap + sense_comb_finger_h)
proof_mass_fingers_pole_w = 15*scale
proof_mass_fingers_pole_h = (3*(sense_comb_finger_dist+sense_comb_finger_h))
sense_comb_finger_w = 243*scale
sense_comb_finger_num = 3
drive_comb_finger_w = 48*scale
drive_comb_finger_h = 9*scale
drive_comb_finger_dist = (small_gap + large_gap + drive_comb_finger_h)
drive_comb_finger_num = 8
mirror_quarter = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w
mirror_gyro = mirror_quarter*2 - drive_anchor_width/2
drive_coupling_beam_w = 118.5*scale
drive_coupling_beam_h = 21*scale
drive_coupling_dist = 42.75*scale
drive_coupling_beam_vert_w = 9*scale
drive_coupling_dist2 = 95*scale
drive_coupling_beam_vert_h = drive_coupling_dist2*2 + drive_coupling_beam_h
drive_coupling_connector_w = 15*scale
drive_coupling_connector_h = 12*scale

Volume = ((drive_anchor_height*drive_anchor_width)*4 + (drive_serpentine_connector_h*drive_serpentine_connector_w)*4
```

```
+ (serpentine_width*drive_serpentine_beam_h)*8 + (drive_serpentine_connector2_h*drive_serpentine_connector2_w)*8 +
(drive_serpentine_connector3_h*drive_serpentine_connector3_w)*4 + (drive_frame_beam_w*drive_frame_beam_h)*4 +
(drive_frame_connector_h*drive_frame_connector_w)*4 + (drive_frame_serpent_beam_w*serpentine_width)*8 +
(drive_frame_base_w*drive_frame_base_h)*4 + (proof_mass_beam_w*proof_mass_beam_h)*4 +
(proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
(sense_comb_finger_h*sense_comb_finger_w)*12 + (drive_comb_finger_h*drive_comb_finger_w)*32)*thickness

Volume_proof_mass = ((proof_mass_beam_w*proof_mass_beam_h)*4 +
(proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
(sense_comb_finger_h*sense_comb_finger_w)*12)*thickness

Volume_drive_frame = ((drive_frame_beam_w*drive_frame_beam_h)*4 + (drive_frame_connector_w*drive_frame_connector_h)*4 +
(drive_frame_serpent_beam_w*serpentine_width)*8 + (drive_serpentine_connector2_w*drive_serpentine_connector2_h)*4 +
(drive_frame_base_h*drive_frame_base_w)*4)*thickness

Volume_drive = Volume_proof_mass + Volume_drive_frame

# Constants
epsilon0 = 8.85e-12
L = 18*scale
Vdc = 8
Vac = 4
drive_frequency, sense_frequency = gyro_modal.main()
mu = 1.86e-5
lamb = 0.067e-6

def drive_amplitude(drive_frequency,q_factor):
    drive_mass = Volume_drive*rho
    kd = (drive_mass * (drive_frequency*2*math.pi)**2)
    f_actuation = 2*epsilon0*L*thickness*drive_comb_finger_num*2*Vdc*Vac*(1/(small_gap**2))
    X0 = q_factor * f_actuation / (drive_mass * (drive_frequency*2*math.pi)**2)
    return X0

def coriolis_force(angular_rate,drive_frequency):
```

```
        mass_coriolis = Volume_drive*rho
        X0 = drive_amplitude()
        F_coriolis = -2*mass_coriolis*angular_rate*X0*drive_frequency*2*math.pi
        return F_coriolis


    def sense_disp(angular_rate,Q_factor,q_factor_drive,drive_frequency, sense_frequency):
        Y0 = angular_rate * ((Volume_drive*rho) * drive_frequency*2*math.pi) * (1/(Volume_proof_mass*rho * (sense_frequency*2*math.pi)**2)) * 2 * drive_amplitude(drive_frequency,q_factor_drive) * (1/math.sqrt ((1-((drive_frequency*2*math.pi)/(sense_frequency*2*math.pi))**2) **2) + (1/Q_factor * ((drive_frequency*2*math.pi)/(sense_frequency *2*math.pi)))**2)
        return Y0

    return sense_disp(1,q_factor_sense,q_factor_drive,drive_frequency, sense_frequency)

# Mesh
mesh = Mesh('gyroscope.xml')

def disp_fem(force,serpentine_width,proof_mass_beam_w,proof_mass_beam_h,
proof_mass_w,proof_mass_h,sense_comb_finger_h):

    scale = 1e-6
    cl = 80*scale
    small_cl = 9*scale
    thickness = 50*scale
    small_gap = 3*scale
    large_gap = 4*small_gap
    drive_anchor_width = 124*scale
    drive_anchor_height = 126*scale
    drive_serpentine_connector_w = 21*scale
    drive_serpentine_connector_h = 24*scale
    drive_serpentine_beam_h = 194*scale
    drive_serpentine_connector2_w = 17*scale
    drive_serpentine_connector2_h = 21*scale
    drive_serpentine_beam2_x = drive_anchor_width + drive_serpentine_connector_w + serpentine_width + drive_serpentine_connector2_w
    drive_serpentine_connector3_w = 17*scale
    drive_serpentine_connector3_h = 24*scale
    drive_frame_beam_w = 56*scale
    drive_frame_beam_h = 485*scale
    drive_frame_connector_w = 72*scale
```

```
drive_frame_connector_h = 73*scale
drive_frame_serpent_beam_w = 171*scale
drive_frame_serpent_connector_x = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + drive_frame_serpent_beam_w - drive_serpentine_connector2_h
drive_frame_base_w = 309*scale
drive_frame_base_h = 41*scale
sense_comb_finger_dist = (small_gap + large_gap + sense_comb_finger_h)
proof_mass_fingers_pole_w = 15*scale
proof_mass_fingers_pole_h = (3*(sense_comb_finger_dist+sense_comb_finger_h))
sense_comb_finger_w = 243*scale
sense_comb_finger_num = 3
drive_comb_finger_w = 48*scale
drive_comb_finger_h = 9*scale
drive_comb_finger_dist = (small_gap + large_gap + drive_comb_finger_h)
drive_comb_finger_num = 8
mirror_quarter = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w
mirror_gyro = mirror_quarter*2 - drive_anchor_width/2
drive_coupling_beam_w = 118.5*scale
drive_coupling_beam_h = 21*scale
drive_coupling_dist = 42.75*scale
drive_coupling_beam_vert_w = 9*scale
drive_coupling_dist2 = 95*scale
drive_coupling_beam_vert_h = drive_coupling_dist2*2 + drive_coupling_beam_h
drive_coupling_connector_w = 15*scale
drive_coupling_connector_h = 12*scale

Volume = ((drive_anchor_height*drive_anchor_width)*4 + (drive_serpentine_connector_h*drive_serpentine_connector_w)*4
+ (serpentine_width*drive_serpentine_beam_h)*8 + (drive_serpentine_connector2_h*drive_serpentine_connector2_w)*8 +
(drive_serpentine_connector3_h*drive_serpentine_connector3_w)*4 + (drive_frame_beam_w*drive_frame_beam_h)*4 +
(drive_frame_connector_h*drive_frame_connector_w)*4 + (drive_frame_serpent_beam_w*serpentine_width)*8 +
(drive_frame_base_w*drive_frame_base_h)*4 + (proof_mass_beam_w*proof_mass_beam_h)*4 +
(proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
(sense_comb_finger_h*sense_comb_finger_w)*12 + (drive_comb_finger_h*drive_comb_finger_w)*32)*thickness

```

```
Volume_proof_mass = ((proof_mass_beam_w*proof_mass_beam_h)*4 +
(proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
(sense_comb_finger_h*sense_comb_finger_w)*12)*thickness

Volume_drive_frame = ((drive_frame_beam_w*drive_frame_beam_h)*4 + (drive_frame_connector_w*drive_frame_connector_h)*4 +
(drive_frame_serpent_beam_w*serpentine_width)*8 + (drive_serpentine_connector2_w*drive_serpentine_connector2_h)*4 +
(drive_frame_base_h*drive_frame_base_w)*4)*thickness

Volume_drive = Volume_proof_mass + Volume_drive_frame
# Strain operator
def eps(v):
    return sym(grad(v))

# Stress tensor
def sigma(v):
    return lmbda*tr(eps(v))*Identity(3) + 2.0*mu*eps(v)

# Function Space
V = VectorFunctionSpace(mesh,'Lagrange',degree=3)
u_ = TrialFunction(V)
du = TestFunction(V)


# Boundary Conditions
# Upper Left Quarter

def drive_anchor_left(x,on_boundary):
    return near(x[0],0.)

def drive_anchor_left2(x,on_boundary):
    return near(x[0],drive_anchor_width)

def drive_anchor_left3(x,on_boundary):
    return near(x[1],0.) and x[0] >= 0 and x[0] <= drive_anchor_width

def drive_anchor_left4(x,on_boundary):
    return near(x[1],drive_anchor_height) and x[0] >= 0 and x[0] <= drive_anchor_width

def drive_anchor_left5(x,on_boundary):
    return near(x[1],mirror_half_helper(drive_frame_beam_h,0)) and x[0] >= 0 and x[0] <= drive_anchor_width
```

```
def drive_anchor_left6(x,on_boundary):
    return near(x[1],mirror_half_helper(drive_frame_beam_h,0)-drive_anchor_height) and x[0] >= 0 and x[0] <= drive_anchor_width

def drive_middle_anchor(x,on_boundary):
    return near(x[0],mirror_quarter_helper(mirror_quarter,0)) and x[1] >= 0 and x[1] <= drive_anchor_height

def drive_middle_anchor2(x,on_boundary):
    return near(x[0],mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width) and x[1] >= 0 and x[1] <= drive_anchor_height

def drive_middle_anchor12(x,on_boundary):
    return near(x[0],mirror_quarter_helper(mirror_quarter,0)) and x[1] >= mirror_half_helper(drive_frame_beam_h,0)-drive_anchor_height and x[1] <= mirror_half_helper(drive_frame_beam_h,0)

def drive_middle_anchor22(x,on_boundary):
    return near(x[0],mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width) and x[1] >= mirror_half_helper(drive_frame_beam_h,0)-drive_anchor_height and x[1] <= mirror_half_helper(drive_frame_beam_h,0)

def drive_middle_anchor3(x,on_boundary):
    return near(x[1],0.) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)

def drive_middle_anchor4(x,on_boundary):
    return near(x[1],drive_anchor_height) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)

def drive_middle_anchor5(x,on_boundary):
    return near(x[1],mirror_half_helper(drive_frame_beam_h,0)) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)

def drive_middle_anchor6(x,on_boundary):
    return near(x[1],mirror_half_helper(drive_frame_beam_h,0)-drive_anchor_height) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)
```

```
    bc = [DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left),
          DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left2),
          DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left3),
          DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left4),
          DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left5),
          DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left6),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor2),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor12),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor22),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor3),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor4),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor5),
          DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor6)]

    # Change the force to x and y, depending on whether the desired displacement is from actuation force or coriolis force (a change in the volume is also need)
    f = Constant((0.,force/Volume_proof_mass,0.))
    T = Constant((0,0,0))
    V = VectorFunctionSpace(mesh,'Lagrange',degree=3)
    du = TrialFunction(V)
    u_ = TestFunction(V)
    a = inner(sigma(du),eps(u_))*dx
    l = dot(f,u_)*dx

    u = Function(V, name='Displacement')
    solve(a == l, u, bc)
    disp = u(mirror_quarter,mirror_half_helper,thickness/2)

    # Set up file for exporting results
    file_results = XDMFFile("gyro_displacement.xdmf")
    file_results.parameters["flush_output"] = True
    file_results.parameters["functions_share_mesh"] = True
    file_results.write(u,0)

    return disp[1]
```

Listing II.3: Modal analysis simulation script

```
# MEMS Gyroscope modal analysis script
# @ruiesteves

# Imports
from fenics import *
import numpy as np
```

```
import time
import math


# Definitions


# For PolySi
E = Constant(170e9)
nu = Constant(0.28)
rho = 2329
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)


# Meshing
mesh = Mesh("gyroscope.xml")


# Helper functions
def mirror_quarter_helper(mirror_quarter,x_point):
    dist = mirror_quarter - x_point
    x_new = mirror_quarter + dist
    return x_new

def mirror_half_helper(drive_frame_beam_h,y_point):
    dist = drive_frame_beam_h - y_point
    y_new = drive_frame_beam_h + dist
    return y_new




def main(serpentine_width,proof_mass_beam_w,proof_mass_beam_h,
proof_mass_w,proof_mass_h,sense_comb_finger_h):

    scale = 1e-6
    cl = 80*scale
    small_cl = 9*scale
    thickness = 50*scale
    small_gap = 3*scale
    large_gap = 4*small_gap
    drive_anchor_width = 124*scale
    drive_anchor_height = 126*scale
```

```
drive_serpentine_connector_w = 21*scale
drive_serpentine_connector_h = 24*scale
drive_serpentine_beam_h = 194*scale
drive_serpentine_connector2_w = 17*scale
drive_serpentine_connector2_h = 21*scale
drive_serpentine_beam2_x = drive_anchor_width + drive_serpentine_connector_w + serpentine_width + drive_serpentine_connector2_w
drive_serpentine_connector3_w = 17*scale
drive_serpentine_connector3_h = 24*scale
drive_frame_beam_w = 56*scale
drive_frame_beam_h = 485*scale
drive_frame_connector_w = 72*scale
drive_frame_connector_h = 73*scale
drive_frame_serpent_beam_w = 171*scale
drive_frame_serpent_connector_x = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + drive_frame_serpent_beam_w - drive_serpentine_connector2_h
drive_frame_base_w = 309*scale
drive_frame_base_h = 41*scale
sense_comb_finger_dist = (small_gap + large_gap + sense_comb_finger_h)
proof_mass_fingers_pole_w = 15*scale
proof_mass_fingers_pole_h = (3*(sense_comb_finger_dist+sense_comb_finger_h))
sense_comb_finger_w = 243*scale
sense_comb_finger_num = 3
drive_comb_finger_w = 48*scale
drive_comb_finger_h = 9*scale
drive_comb_finger_dist = (small_gap + large_gap + drive_comb_finger_h)
drive_comb_finger_num = 8
mirror_quarter = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w
mirror_gyro = mirror_quarter*2 - drive_anchor_width/2
drive_coupling_beam_w = 118.5*scale
drive_coupling_beam_h = 21*scale
drive_coupling_dist = 42.75*scale
drive_coupling_beam_vert_w = 9*scale
drive_coupling_dist2 = 95*scale
drive_coupling_beam_vert_h = drive_coupling_dist2*2 + drive_coupling_beam_h
drive_coupling_connector_w = 15*scale
drive_coupling_connector_h = 12*scale

# Functions
def eps(v):
    #return 0.5*(nabla_grad(v) + nabla_grad(v).T)
```

```python
        return sym(grad(v))

    def sigma(v):
        return lmbda*tr(eps(v))*Identity(3) + 2.0*mu*eps(v)

    # Function Space
    V = VectorFunctionSpace(mesh,'Lagrange',degree=3)
    u_ = TrialFunction(V)
    du = TestFunction(V)

    # Boundary Conditions
    # Upper Left Quarter

    def drive_anchor_left(x,on_boundary):
        return near(x[0],0.)

    def drive_anchor_left2(x,on_boundary):
        return near(x[0],drive_anchor_width)

    def drive_anchor_left3(x,on_boundary):
        return near(x[1],0.) and x[0] >= 0 and x[0] <= drive_anchor_width

    def drive_anchor_left4(x,on_boundary):
        return near(x[1],drive_anchor_height) and x[0] >= 0 and x[0] <=
            ↪ drive_anchor_width

    def drive_anchor_left5(x,on_boundary):
        return near(x[1],mirror_half_helper(drive_frame_beam_h,0)) and x[0] >= 0
            ↪ and x[0] <= drive_anchor_width

    def drive_anchor_left6(x,on_boundary):
        return near(x[1],mirror_half_helper(drive_frame_beam_h,0)-
            ↪ drive_anchor_height) and x[0] >= 0 and x[0] <= drive_anchor_width

    def drive_middle_anchor(x,on_boundary):
        return near(x[0],mirror_quarter_helper(mirror_quarter,0)) and x[1] >= 0
            ↪ and x[1] <= drive_anchor_height

    def drive_middle_anchor2(x,on_boundary):
        return near(x[0],mirror_quarter_helper(mirror_quarter,0)-
            ↪ drive_anchor_width) and x[1] >= 0 and x[1] <= drive_anchor_height

    def drive_middle_anchor12(x,on_boundary):
```

```
    return near(x[0],mirror_quarter_helper(mirror_quarter,0)) and x[1] >= mirror_half_helper(drive_frame_beam_h,0)-drive_anchor_height and x[1] <= mirror_half_helper(drive_frame_beam_h,0)

def drive_middle_anchor22(x,on_boundary):
    return near(x[0],mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width) and x[1] >= mirror_half_helper(drive_frame_beam_h,0)-drive_anchor_height and x[1] <= mirror_half_helper(drive_frame_beam_h,0)

def drive_middle_anchor3(x,on_boundary):
    return near(x[1],0.) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)

def drive_middle_anchor4(x,on_boundary):
    return near(x[1],drive_anchor_height) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)

def drive_middle_anchor5(x,on_boundary):
    return near(x[1],mirror_half_helper(drive_frame_beam_h,0)) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)

def drive_middle_anchor6(x,on_boundary):
    return near(x[1],mirror_half_helper(drive_frame_beam_h,0)-drive_anchor_height) and x[0] >= mirror_quarter_helper(mirror_quarter,0)-drive_anchor_width and x[0] <= mirror_quarter_helper(mirror_quarter,0)





bc = [DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left),
      DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left2),
      DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left3),
      DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left4),
      DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left5),
      DirichletBC(V, Constant((0.,0.,0.)),drive_anchor_left6),
      DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor),
      DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor2),
      DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor12),
      DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor22),
```

```
     DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor3),
     DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor4),
     DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor5),
     DirichletBC(V, Constant((0.,0.,0.)),drive_middle_anchor6)]


# Matrices
k_form = inner(sigma(du),eps(u_))*dx
l_form = Constant(1.)*u_[0]*dx
K = PETScMatrix()
b = PETScVector()
assemble_system(k_form,l_form,bc,A_tensor=K,b_tensor=b)

m_form = rho*dot(du,u_)*dx
M = PETScMatrix()
assemble(m_form, tensor=M)

# Eigenvalues/Eigensolver
eigensolver = SLEPcEigenSolver(K,M)
eigensolver.parameters['problem_type'] = 'gen_hermitian'
#eigensolver.parameters['spectrum'] = 'smallest real'
eigensolver.parameters['spectral_transform'] = 'shift-and-invert'
eigensolver.parameters['spectral_shift'] = 0.
#PETScOptions.set("st_pc_factor_mat_solver_type", "mumps")
N_eig = 6
eigensolver.solve(N_eig)
#print (eigensolver.parameters.str(True))

# Export results
file_results = XDMFFile('gyro_modal_analysis.xdmf')
file_results.parameters['flush_output'] = True
file_results.parameters['functions_share_mesh'] = True

r1,c1,rx1,cx1 = eigensolver.get_eigenpair(0)
r3,c3,rx3,cx3 = eigensolver.get_eigenpair(3)
u = Function(V)
u.vector()[:] = rx1
file_results.write(u,0)

# Extraction
for i in range(N_eig):
    r,c,rx,cx = eigensolver.get_eigenpair(i)
    freq = sqrt(r)/2/pi
    print('Mode:',i,'   ','Freq:',freq,'[Hz]')
```

```
    freq_final_drive = sqrt(r1)/2/pi
    freq_final_sense = sqrt(r3)/2/pi

    return freq_final_drive, freq_final_sense
```

Listing II.4: Damping calculation script

```
# MEMS Gyroscope squeeze and slide film damping calculation script
# @ruiesteves

# Imports
import gyro_modal
import math

# Constants
scale = 1e-6
L = 18*scale
small_gap = 3*scale
mu = 1.86e-5
lamb = 0.067e-6
thickness = 50*scale
drive_comb_finger_num = 8

# Slide damping calculation
def damping_drive():
    Kn = lamb/small_gap
    mu_eff = mu/(1 + 2*Kn + (0.2*Kn**0.788)*math.exp(-Kn/10))
    A = L*thickness
    c_drive = 4 * drive_comb_finger_num * mu_eff * (A/small_gap)
    return c_drive

def q_factor_drive(serpentine_width,proof_mass_beam_w,proof_mass_beam_h,
    ↪ proof_mass_w,proof_mass_h,sense_comb_finger_h,drive_frequency):
    # Constants
    scale = 1e-6
    cl = 80*scale
    small_cl = 9*scale
    thickness = 50*scale
    small_gap = 3*scale
    large_gap = 4*small_gap
    drive_anchor_width = 124*scale
    drive_anchor_height = 126*scale
    drive_serpentine_connector_w = 21*scale
    drive_serpentine_connector_h = 24*scale
    drive_serpentine_beam_h = 194*scale
    drive_serpentine_connector2_w = 17*scale
```

```
drive_serpentine_connector2_h = 21*scale
drive_serpentine_beam2_x = drive_anchor_width + drive_serpentine_connector_w + serpentine_width + drive_serpentine_connector2_w
drive_serpentine_connector3_w = 17*scale
drive_serpentine_connector3_h = 24*scale
drive_frame_beam_w = 56*scale
drive_frame_beam_h = 485*scale
drive_frame_connector_w = 72*scale
drive_frame_connector_h = 73*scale
drive_frame_serpent_beam_w = 171*scale
drive_frame_serpent_connector_x = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + drive_frame_serpent_beam_w - drive_serpentine_connector2_h
drive_frame_base_w = 309*scale
drive_frame_base_h = 41*scale
sense_comb_finger_dist = (small_gap + large_gap + sense_comb_finger_h)
proof_mass_fingers_pole_w = 15*scale
proof_mass_fingers_pole_h = (3*(sense_comb_finger_dist+sense_comb_finger_h))
sense_comb_finger_w = 243*scale
sense_comb_finger_num = 3
drive_comb_finger_w = 48*scale
drive_comb_finger_h = 9*scale
drive_comb_finger_dist = (small_gap + large_gap + drive_comb_finger_h)
drive_comb_finger_num = 8
mirror_quarter = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w
mirror_gyro = mirror_quarter*2 - drive_anchor_width/2
drive_coupling_beam_w = 118.5*scale
drive_coupling_beam_h = 21*scale
drive_coupling_dist = 42.75*scale
drive_coupling_beam_vert_w = 9*scale
drive_coupling_dist2 = 95*scale
drive_coupling_beam_vert_h = drive_coupling_dist2*2 + drive_coupling_beam_h
drive_coupling_connector_w = 15*scale
drive_coupling_connector_h = 12*scale

Volume = ((drive_anchor_height*drive_anchor_width)*4 + ( drive_serpentine_connector_h*drive_serpentine_connector_w)*4
+ (serpentine_width*drive_serpentine_beam_h)*8 + ( drive_serpentine_connector2_h*drive_serpentine_connector2_w)*8 +
(drive_serpentine_connector3_h*drive_serpentine_connector3_w)*4 + ( drive_frame_beam_w*drive_frame_beam_h)*4 +
```

```
    (drive_frame_connector_h*drive_frame_connector_w)*4 + (drive_frame_serpent_beam_w*serpentine_width)*8 +
    (drive_frame_base_w*drive_frame_base_h)*4 + (proof_mass_beam_w*proof_mass_beam_h)*4 +
    (proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
    (sense_comb_finger_h*sense_comb_finger_w)*12 + (drive_comb_finger_h*drive_comb_finger_w)*32)*thickness

    Volume_proof_mass = ((proof_mass_beam_w*proof_mass_beam_h)*4 +
    (proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
    (sense_comb_finger_h*sense_comb_finger_w)*12)*thickness

    Volume_drive_frame = ((drive_frame_beam_w*drive_frame_beam_h)*4 + (drive_frame_connector_w*drive_frame_connector_h)*4 +
    (drive_frame_serpent_beam_w*serpentine_width)*8 + (drive_serpentine_connector2_w*drive_serpentine_connector2_h)*4 +
    (drive_frame_base_h*drive_frame_base_w)*4)*thickness

    Volume_drive = Volume_proof_mass + Volume_drive_frame

    drive_mass = Volume_drive*rho
    c_drive = damping_drive()
    q_factor = drive_mass * (drive_frequency*2*math.pi) / c_drive
    return q_factor

# Squeeze film damping calculation
def damping_sense():
    Kn = lamb/small_gap
    mu_eff = mu/(1+9.638*Kn**1.159)
    Pa = 101.3e3
    A = L_sense*thickness
    c = L_sense/thickness
    squeeze_number = (12*mu_eff*10*2*math.pi*L_sense**2)/(Pa*small_gap**2)
    sum = 0
    for m in range(1,10,2):
        for n in (1,10,2):
            sum = sum + (m**2 + c**2 * n**2)/((m*n)**2 * ((m**2 + c**2 * n**2)**2 + (squeeze_number**2 / math.pi**4)))

    F_damping = ((64*squeeze_number*Pa*A)/(math.pi**6 * small_gap)) * sum
    c_sense = F_damping * sense_comb_finger_num * 4
    return c_sense

```

```python
def q_factor_sense(serpentine_width,proof_mass_beam_w,proof_mass_beam_h,proof_mass_w,proof_mass_h,sense_comb_finger_h,sense_frequency):
    # Constants
    scale = 1e-6
    cl = 80*scale
    small_cl = 9*scale
    thickness = 50*scale
    small_gap = 3*scale
    large_gap = 4*small_gap
    drive_anchor_width = 124*scale
    drive_anchor_height = 126*scale
    drive_serpentine_connector_w = 21*scale
    drive_serpentine_connector_h = 24*scale
    drive_serpentine_beam_h = 194*scale
    drive_serpentine_connector2_w = 17*scale
    drive_serpentine_connector2_h = 21*scale
    drive_serpentine_beam2_x = drive_anchor_width + drive_serpentine_connector_w + serpentine_width + drive_serpentine_connector2_w
    drive_serpentine_connector3_w = 17*scale
    drive_serpentine_connector3_h = 24*scale
    drive_frame_beam_w = 56*scale
    drive_frame_beam_h = 485*scale
    drive_frame_connector_w = 72*scale
    drive_frame_connector_h = 73*scale
    drive_frame_serpent_beam_w = 171*scale
    drive_frame_serpent_connector_x = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + drive_frame_serpent_beam_w - drive_serpentine_connector2_h
    drive_frame_base_w = 309*scale
    drive_frame_base_h = 41*scale
    sense_comb_finger_dist = (small_gap + large_gap + sense_comb_finger_h)
    proof_mass_fingers_pole_w = 15*scale
    proof_mass_fingers_pole_h = (3*(sense_comb_finger_dist+sense_comb_finger_h))
    sense_comb_finger_w = 243*scale
    sense_comb_finger_num = 3
    drive_comb_finger_w = 48*scale
    drive_comb_finger_h = 9*scale
    drive_comb_finger_dist = (small_gap + large_gap + drive_comb_finger_h)
    drive_comb_finger_num = 8
    mirror_quarter = drive_serpentine_beam2_x + serpentine_width + drive_serpentine_connector3_w+drive_frame_beam_w + drive_frame_connector_w + proof_mass_w
    mirror_gyro = mirror_quarter*2 - drive_anchor_width/2
    drive_coupling_beam_w = 118.5*scale
```

```
    drive_coupling_beam_h = 21*scale
    drive_coupling_dist = 42.75*scale
    drive_coupling_beam_vert_w = 9*scale
    drive_coupling_dist2 = 95*scale
    drive_coupling_beam_vert_h = drive_coupling_dist2*2 + drive_coupling_beam_h
    drive_coupling_connector_w = 15*scale
    drive_coupling_connector_h = 12*scale

    Volume = ((drive_anchor_height*drive_anchor_width)*4 + (drive_serpentine_connector_h*drive_serpentine_connector_w)*4
    + (serpentine_width*drive_serpentine_beam_h)*8 + (drive_serpentine_connector2_h*drive_serpentine_connector2_w)*8 +
    (drive_serpentine_connector3_h*drive_serpentine_connector3_w)*4 + (drive_frame_beam_w*drive_frame_beam_h)*4 +
    (drive_frame_connector_h*drive_frame_connector_w)*4 + (drive_frame_serpent_beam_w*serpentine_width)*8 +
    (drive_frame_base_w*drive_frame_base_h)*4 + (proof_mass_beam_w*proof_mass_beam_h)*4 +
    (proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
    (sense_comb_finger_h*sense_comb_finger_w)*12 + (drive_comb_finger_h*drive_comb_finger_w)*32)*thickness

    Volume_proof_mass = ((proof_mass_beam_w*proof_mass_beam_h)*4 +
    (proof_mass_w*proof_mass_h)*4 + (proof_mass_fingers_pole_w*proof_mass_fingers_pole_h)*4 +
    (sense_comb_finger_h*sense_comb_finger_w)*12)*thickness

    Volume_drive_frame = ((drive_frame_beam_w*drive_frame_beam_h)*4 + (drive_frame_connector_w*drive_frame_connector_h)*4 +
    (drive_frame_serpent_beam_w*serpentine_width)*8 + (drive_serpentine_connector2_w*drive_serpentine_connector2_h)*4 +
    (drive_frame_base_h*drive_frame_base_w)*4)*thickness

    Volume_drive = Volume_proof_mass + Volume_drive_frame

    mass_sense = Volume_proof_mass*rho
    c_sense = damping_sense()
    q_factor = mass_sense * sense_frequency*2*math.pi / c_sense
    return q_factor
```

Listing II.5: Electrical domain simulation script

```
# MEMS Gyroscope electrical domain simulation
# @ruiesteves

```

```
# Imports
import gyro_disp

# Constants
scale = 1e-6
thickness = 50*scale
sense_comb_finger_num = 3
small_gap = 3*scale
L_sense = 18*scale
epsilon0 = 8.85e-12

# Functions
def main(serpentine_width,proof_mass_beam_w,proof_mass_beam_h,proof_mass_w,proof_mass_h,sense_comb_finger_h,q_factor_sense,q_factor_drive,drive_frequency,sense_frequency):
    disp = gyro_disp.disp_equations(serpentine_width,proof_mass_beam_w,proof_mass_beam_h,proof_mass_w,proof_mass_h,sense_comb_finger_h,q_factor_sense,q_factor_drive,drive_frequency,sense_frequency)

    def cap_change(disp):
        C = 2*sense_comb_finger_num*2 * epsilon0 * thickness * L_sense * disp / (small_gap**2)
        print("Cap change:",C*1e15,"fF")
        return C

    def c2v(cap):
        v = 2 * cap * 2.5 * (1/100e-15)
        print("Output voltage:",v*1e3,"mV")
        return v

    return c2v(cap_change(disp))
```

Listing II.6: Genetic algorithm script for MEMS gyroscope

```
# Python Gyroscope GA
# @ruiesteves

# Imports
import gyro_geo
import gyro_disp
import gyro_elec
import gyro_modal
import gyro_damping
import numpy as np
import math as math
import random as rand
```

```
import copy

# Initial parameters of the device to be optimized
scale = 1e-6
suspension_beam_width = 20*scale
proof_mass_frame_width = 430*scale
proof_mass_frame_length = 60*scale
proof_mass_width = 290*scale
proof_mass_length = 220*scale
sense_comb_finger_w = 14*scale

initial = [suspension_beam_width,proof_mass_frame_width,proof_mass_frame_length,
    proof_mass_width,proof_mass_length,sense_comb_finger_w]

# Classes
class GA_device:

    def __init__(self,id):
        self.list_parameters = []
        self.id = id

    def calc_freq(self):
        list = self.list_parameters
        self.freq_drive, self.freq_sense = gyro_modal.main(list[0],list[1],list
            [2],list[3],list[4],list[5])

    def calc_qfactor(self):
        list = self.list_parameters
        self.qfactor_drive = gyro_damping.q_factor_drive(list[0],list[1],list[2],
            list[3],list[4],list[5],self.freq_drive)
        self.qfactor_sense = gyro_damping.q_factor_sense(list[0],list[1],list[2],
            list[3],list[4],list[5],self.freq_sense)

    def calc_sensitivity(self):
        list = self.list_parameters
        self.sensitivity = gyro_elec.main(list[0],list[1],list[2],list[3],list[4],
            list[5],self.q_factor_drive,self.q_factor_sense,self.freq_drive,
            self.freq_sense)

    def calc_fom(self):
        list = self.list_parameters
        try:
            gyro_geo.build(list[0],list[1],list[2],list[3],list[4],list[5])
            self.calc_freq()
            self.calc_qfactor()
```

```
            self.calc_sensitivity()
            self.fom = (1/(self.freq_sense - self.freq_drive) * self.sensitivity * self.qfactor_sense * 1e6
        except:
            print("Geometry became invalid for device",self.id)
            self.fom = 0


class GA: # GA class, initiated with a list of devices, a list of mutation chances and a list of mutation relative size

    def __init__(self,list_devices,mutation_chance,mutation_size):
        self.list_devices = list_devices # Must be a list of GA_devices
        self.mutation_chance = mutation_chance # A list, with different (or not) mutation chances for each parameter
        self.mutation_size = mutation_size # Same as above, this time for mutation_sizes (IMPORTANT to check)

    def mutate(self,dev): # The mutation function, mutating the parameters according to their mutation chance and size
        le = len(dev.list_parameters)
        for i in range(le):
            if rand.uniform(0,1) < self.mutation_chance[i]:
                if rand.uniform(0,1) < 0.5:
                    dev.list_parameters[i] = dev.list_parameters[i] + dev.list_parameters[i]*self.mutation_size[i]
                else:
                    dev.list_parameters[i] = dev.list_parameters[i] - dev.list_parameters[i]*self.mutation_size[i]

    def reproduce(self,top_25):
        new_population = []

        for dev in top_25: # Passing the best 25 devices to the next generation
            new_population.append(dev)

        for dev in top_25: # Copying and mutating the best 25 devices to the next generation
            new_dev = copy.deepcopy(dev)
            self.mutate(new_dev)
            new_population.append(new_dev)

        for i in range(len(self.list_devices)//2): # Randomly mutating and passing half of the population to the next generation
            new_dev_r = copy.deepcopy(self.list_devices[i])
```

```
            self.mutate(new_dev_r)
            new_population.append(new_dev_r)

        return new_population


    def one_generation(self):

        for dev in self.list_devices:
            dev.calc_fom()

        le = len(self.list_devices)
        scores = [self.list_devices[i].fom for i in range(le)]
        max = np.amax(scores)
        print(scores)
        print(max)

        top_25_index = list(np.argsort(scores))[3*(le//4):le]
        top_25 = [self.list_devices[i] for i in top_25_index][::-1]

        self.list_devices = self.reproduce(top_25)



# Script
print("Genetic algorithm optimization for MEMS Gyroscope")
num_pop = int(input("Size of the population: "))
num_gen = int(input("Number of generations: "))

initial_pop = []
for i in range(num_pop):
    initial_pop.append(GA_device(i))
    for par in range(len(initial)):
        initial_pop[i].list_parameters.append(initial[par])
    initial_pop[i].calc_fom()

init_ga = GA(initial_pop ,[0.6,0.6,0.6,0.6,0.6,0.6],[0.122,0.014,0.017,0.0105,0.01387,0.065])

for i in range(num_gen):
    init_ga.one_generation()
    for dev in init_ga.list_devices:
        print("\n","For Device number",dev.id,":")
        print(dev.fom,"FOM")

```

```
max_score = 0


for dev in init_ga.list_devices:
    if dev.fom > max_score:
        max_score = dev.fom

print("Maximum FOM:",max_score)
```

Annex III

Permittivity values

Table III.1: Permittivity values

Permittivity	Value
ϵ_0 (free space)	8.85×10^{-12} F/m
ϵ_r (air)	1 F/m