

# PhaRoughLife: A FEniCSx & Gmsh code to predict the effect of surface roughness in the fatigue life of materials.

Sara Jiménez-Alfaro<sup>a,b</sup>, Emilio Martínez-Pañeda<sup>b</sup>,

<sup>a</sup>*Department of Civil and Environmental Engineering, Imperial College London, London SW7 2AZ, UK*

<sup>b</sup>*Department of Engineering Science, University of Oxford, Oxford OX1 3PJ, UK*

---

## Abstract

Documentation that accompanies the FEniCSx + Gmsh code PhaRoughLife. This documentation explains the usage of the implemented finite element framework, and highlight the main files.

If using this module for research or industrial proposals, please cite: Jiménez-Alfaro, S., & Martínez-Pañeda, E. A computational framework for predicting the effect of surface roughness in fatigue. *International Journal of Fatigue* 199, 109044 (2025).

*Keywords:* FEniCSx, Gmsh, Phase field fracture, Fatigue, Roughness, Surface factor, Finite element analysis

---

## 1. Introduction

A particularly critical factor in the fatigue life of materials is the surface finish, due to the stress concentrator factors that promote the generation of cracks at the surface. The influence of surface finish is typically quantified by the so-called surface factor [1], defined as

$$K_s = \frac{\sigma_e^r}{\sigma_e^p}, \quad (1)$$

where  $\sigma_e^r$  and  $\sigma_e^p$  represent the endurance limit of the rough and polished surfaces. Here, we present a numerical framework to estimate this surface factor as a function of key parameters, such as the surface topology or the material properties. The proposed model combines together the implementation in FEniCSx of the phase field approach for fatigue fracture introduced in Ref. [2], together with the stochastic meshing strategy for rough surfaces presented in Ref. [3], and adapted for 2D surfaces in this documentation.

---

*Email address:* [emilio.martinez-paneda@eng.ox.ac.uk](mailto:emilio.martinez-paneda@eng.ox.ac.uk) (Emilio Martínez-Pañeda)

### 1.1. Basic usage

For simulating the model as provided, running the function "main\_PhaRoughLife.py" performs all required actions: It automatically generates the geometry and rough mesh, initialises all simulation components, and save outputs in a generated folder called "output". Simple changes, e.g. editing parameters, can be done in this file without requiring altering other files.

To run this code, it is necessary to have the finite element software FEniCSx installed [4], as well as the mesh generation software Gmsh [5]. A detailed tutorial on how to install both open-source programs can be found at the following link. There are several ways to run this code; however, this documentation recommends using the Docker platform. The version of DOLFINx for which this code is compatible is v0.7.3. Hence, the image recommended is included in this link. Hence, at the folder where "main\_PhaRoughLife.py" is allocated, the following command can be run in the terminal:

```
Docker run -ti --rm --name container_name -v ${pwd}:/root/shared/ dolfinx/dolfinx:v0.7.3
```

The container will be opened and then write

```
cd shared
python3 main_PhaRoughLife.py
```

to start the execution. The code works in parallel with the package MPI.

## 2. Summary of included files

### 2.1. The input file: main\_PhaRoughLife.py

Three parts can be distinguished in this code. In the first part the input parameters are included.

```
#####
# Initial parameters
#####
Parameters_data = { "Load":340., "Gc":18.24, "l0":2.9, "E": 200e3, "nu": 0.3, "model":"AT1", "NCycles":10000, '
    Njump':1, "a":485.9, "b":0.0442, "Ra":0.2, "lcorl":50}
simulation_vector = np.linspace(start = 1, stop = 5, num = 5)
rmsr = 1.25*Parameters_data.get('Ra')/1000
Parameters_mesh = {"m0":min(Parameters_data.get("lcorl")/1000, Parameters_data.get("l0"))/5, "m1":1.0, "
    m2":0.05, "n":20, "d":0.8}
```

The dictionary Parameters\_data contains all the parameters that are required in the simulation. They are summarised in Table 1.

Input	Name	Input	Name
Load	Stress amplitude	model	Phase Field model (AT1 or AT2)
Gc	Critical Energy Release Rate	Ncycles	Maximum number of cycles in the simulation
l0	Phase Field length scale	Ra	Average surface roughness (in $\mu\text{m}$ )
E	Young's Modulus	lcorl	Correlation length (in $\mu\text{m}$ )
nu	Poisson's ratio ( $\nu$ )	a,b	Parameters in the Basquin's law $\sigma_a = aN_f^{-b}$
Njump	Number of cycles after which a result is saved in the output file.		

Table 1: Inputs in the code

The simulation numpy vector indicates the number of simulations to be added to the sample. As explained in [6], each time this code is executed, a new rough profile is generated, and consequently, a new estimation of the number of cycles to failure,  $N_f$ , is obtained using the Phase Field model. For example, only 1 rough profiles were generated. The dictionary Parameters\_mesh contains parameters related to the mesh and depends on the geometry of the problem that the user wants to solve. More details are provided below. It can be observed that the minimum mesh size ( $m_0$ ) is always defined according to the rule  $5m_0 = \min(\ell, \ell_{\text{cor}})$ .

The second part involves the iteration procedure, where  $N_f$  is calculated as many times as specified in the simulation vector. Typically, at least 30 simulations are recommended to ensure that the sample follows a Gaussian distribution. For each simulation, the generated data is stored in a folder named output, with filenames clearly identifying the main parameters: the correlation length, the average surface roughness, and the simulation index. The rough profile is created by calling the function roughmesh.py, located in the folder mesh\_generation (inside the folder phasefield\_pycodes). The outputs of this function are the mesh, the cells, and the facets, along with a point (coords\_point) used to apply a fixed boundary condition in the problem. Then, the Phase Field model is applied using the function pfsolver\_fatigue.py. The outputs of this function are saved directly in the folder.

The third part is the calculation of the expected value of the number of cycles to failure, and the surface factor together with the margin of error. An auxiliar function is called to perform all the calculations, name as Ks\_calculator.py.

## 2.2. The execution file: roughmesh.py and its complementary functions

This function first generates the polished mesh (without surface roughness) using the Gmsh function "smoothmesh.py". Then, the function "roughcalcul.py".

### Function to build the nominal surface: smoothmesh.py

The rough mesh is generated following the nominal boundary of the polished surface (without the roughness) using a Gmsh code. Two outputs are needed from the nominal surface (1) The polished mesh in a format .ply2, that will be used to generate the roughness, and (2) the tags related to the external boundary in the nominal surface. The latter is essential to recover the physical groups in the new mesh, that are important to define the boundary conditions in FEniCSx.

(1) The mesh of the polished specimen: The geometry of the specimen corresponds to the one indicated in [6]. In a Gmsh file, four main geometrical elements are needed to generate the mesh: points, lines, loops and surfaces. An example of the commands used to generate them are

```
# Point (the first geometrical point)
p1 = gmsh.model.geo.addPoint(0,0,0,m1)
# Line (Between point p1 and point p2)
l1 = gmsh.model.geo.addLine(p1,p2)
# Loop (lines to generate the surface 1)
cloop1 = gmsh.model.geo.addCurveLoop([l1,l2,l17,l19,l20,l12,l13,l14,l15])
# Surface (made from the first loop)
surface1 = gmsh.model.geo.addPlaneSurface([cloop1],1)
```

(2) The external contour of the nominal surface. In Gmsh, we propose to store the geometrical points and nodes in physical groups which will be later saved as .dat files. Notice that the procedure may depend on the geometry. For example, in this case we have two construction points that are not part of the final geometry. As an example, here we attach the commands used to define the physical group made by the geometrical points (gdim indicates the dimension of the problem, in this case 2D).

```
gmsh.model.addPhysicalGroup(gdim-2, boundpoints,1)
gmsh.model.setPhysicalName(gdim-2, 1, "Boundary_Points")
```

For each of the geometrical boundaries and points, we store the tags generated for the nodes in a vector called nodes\_boundary, which will be later saved as ".dat" file. For example,

```

for tag in gmsh.model.getEntitiesForPhysicalGroup(gdim-2,1):
    nodeTags, _, _ = gmsh.model.mesh.getNodes(gdim-2, tag)
    nodes_boundary.extend(nodeTags-1)

```

### Function to build the rough surface: roughcalcul.py

This part of the code is inspired in the one proposed in Ref. [3], although some changes have been made to adapt it to 2D surfaces. The function import the nodes generated in smoothmesh.py, and then obtain the autocorrelation matrix  $\mathbf{R}$ . The new rough mesh is saved in a .ply2 file. Hence, two functions to read and write the .ply2 format are needed (read\_ply2.py and write\_ply2.py).

The computation of  $\mathbf{R}$  is performed through the generation of the outward normal vector ( $\mathbf{n}$ ), which is computed by the function get\_normal.py. This function is specialized for the 2D case, although a general version for the 3D case is provided in Ref. [3]. From the function smoothmesh.py, we obtain the 2D triangular elements of the polished mesh. Each 2D element is defined by three nodes (1, 2, and 3). If an element lies on the boundary, at least two of its nodes belong to the external contour of the surface, as illustrated in Fig. 1, and are always referred to in the code as coord1 and coord2. These two points are used to calculate the normal vector by defining a local coordinate system  $X'Y'$ . The outward orientation of the normal is determined using the third point (coord3 in the code) together with the local coordinate system.

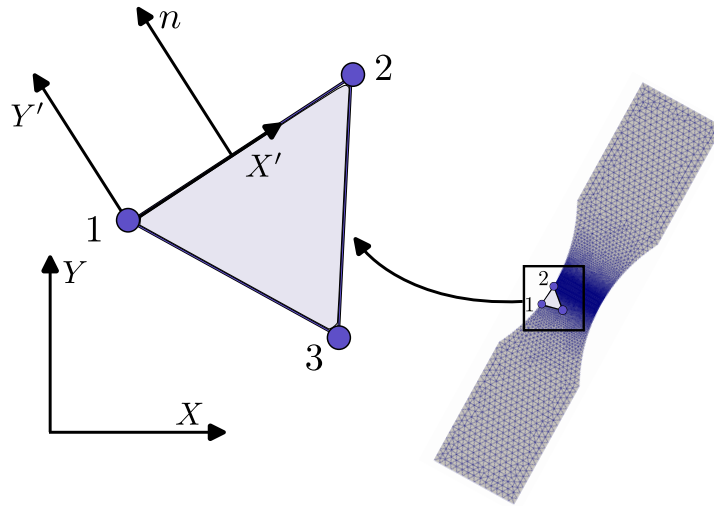


Figure 1: Notation used to obtain the normal vector for an element in the boundary.

### Function to import the roughmesh: importmesh.py

In this function, the nodes generated in the rough mesh are imported as geometrical points to define the mesh used in FEniCSx. The code ensures that physical groups can be defined, which is essential for applying boundary conditions in FEniCSx. The code is entirely dependent on the geometry provided by the user.

### 2.3. The execution file: *pfsolver\_fatigue.py* and its complementary functions

The file *pfsolver\_fatigue.py* contains the execution of the phase field model for fatigue introduced in Ref. [2]. The model works for both AT1 and AT2. A quadrature element is defined for the history variable, since it seems to give more accuracy in the results.

The code builds to SNES solvers for both the damage and the displacement problem. The auxiliary function *SNES\_solver.py* contains more details about the solver. This function is inspired in the NEWFRAC tutorial webpage.

An initial problem is solved to find the endurance energy density ( $\alpha_e$  in [6]). Notice that here only the displacement solver is used, since zero damage is expected at the endurance limit.

```
q.value = petsc4py.PETSc.ScalarType((qe,0.0))
# Displacement problem
u_solver.solve(None, u.vector)
u_solver.destroy, J_u.destroy(), F_u.destroy()
u.vector.ghostUpdate(addv=PETSc.InsertMode.INSERT, mode=PETSc.ScatterMode.FORWARD)
# Update the history variable
a_e.interpolate(dolfinx.fem.Expression(Psip_0(u), V_f.element.interpolation_points()))
a_e.vector.ghostUpdate(addv=PETSc.InsertMode.INSERT, mode=PETSc.ScatterMode.FORWARD)
u.x.array[:] = 0.
ffiled.write_function(a_e)
```

An important point is the definition of the traction force per unit length  $q_e$  that is related to the endurance limit  $\sigma_e$ , the stress located at the center of the specimen, see Fig. 2:  $q_e = \sigma_e \frac{A_i}{A_o}$  where  $A_i$  is the inner surface and  $A_o$  the outer surface.

The problem is solved following a staggered scheme. First, the displacement problem is solved. Then, the fatigue degradation function is updated. Finally, the damage problem is solved. The iteration process finish when there is a convergence in the damage variable between one iteration and the other

```
# Displacement problem
```

```

u_solver.solve(None, u.vector)
u_solver.destroy(), J_u.destroy(), F_u.destroy()
u.vector.ghostUpdate(addv=PETSc.InsertMode.INSERT, mode=PETSc.ScatterMode.FORWARD)

# Update the degradation function
a.interpolate(dolfinx.fem.Expression(a_function(a_old,amax_old,u,a_e), a.function_space.element.
    interpolation_points()))
a.vector.ghostUpdate(addv=PETSc.InsertMode.INSERT, mode=PETSc.ScatterMode.FORWARD)
f.interpolate(dolfinx.fem.Expression(f_function(a), f.function_space.element.interpolation_points()))
f.vector.ghostUpdate(addv=PETSc.InsertMode.INSERT, mode=PETSc.ScatterMode.FORWARD)

# Damage problem
d_solver.solve(None, d.vector)
d_solver.destroy(), J_d.destroy(), F_d.destroy()
d.vector.ghostUpdate(addv=PETSc.InsertMode.INSERT, mode=PETSc.ScatterMode.FORWARD)

```

An important point that saves computation time is that the code stops when the damage variable in the middle point of the specimen is bigger than 0.95. This is achieved in the piece of code

```

if d_global[midpoint_position] >= 0.95:
    # This loop captures when the specimen is broken
    flag_break = 1
    if comm.rank==0:
        Numerics.append([nc,niter,errorL2])
        np.savetxt(str("output/" + str( name_input + '_numerics.dat')), Numerics)

```

The computational time is measured in minutes.

### Function to read the output data: Ks\_calculator.py

A key point in the analysis is to understand the sample, read it, and determine the expected value of the number of cycles to failure. This is the main task of the function Ks\_calculator.py.

### 3. Sample results

The problem shown in Fig. 2 is solved as an example. Table 2 indicates the parameters selected. Notice that the load introduced as an input parameter is the stress amplitude, that is referred to the inner surface  $A_i$ .

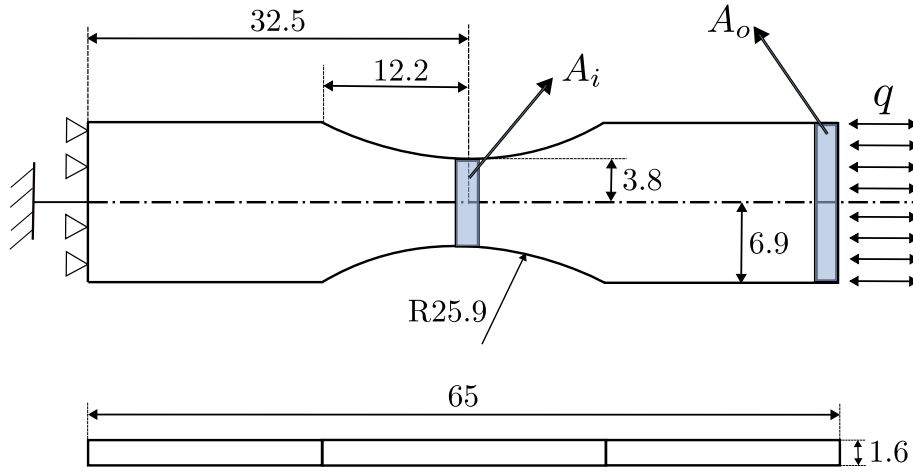


Figure 2: Problem solved in the example.

Input	Name	Input	Name
Load	340 MPa	model	AT1
Gc	18.24 MPa mm	Ncycles	5000
l0	2.9 mm	Ra	1.5 $\mu\text{m}$
E	211 GPa	lcorl	30 in $\mu\text{m}$
nu	0.3	a,b	485.9, 0.0442
Njump	1		

Table 2: Inputs in the code for the example

At the stress concentrators where the fatigue degradation function tends to zero we can observe that the damage variable reaches its critical value 1, as shown in Figs. 3 and 4.

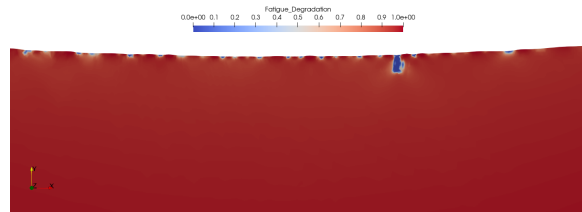


Figure 3: Fatigue degradation function.



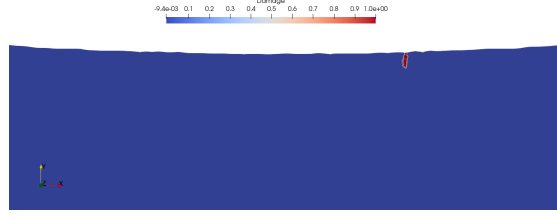


Figure 4: Fatigue degradation function.

## References

- [1] J. Marin, Mechanical behavior of engineering materials, Prentice-Hall, 1962.
- [2] A. Golahmar, C. F. Niordson, E. Martínez-Pañeda, A phase field model for high-cycle fatigue: Total-life analysis, *International Journal of Fatigue* 170 (2023) 107558.
- [3] F. Loth, T. Kiel, K. Busch, P. T. Kristensen, Surface roughness in finite-element meshes: application to plasmonic nanostructures, *Journal of the Optical Society of America B* 40 (3) (2023) B1–B7.
- [4] M. Alnaes, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, G. N. Wells, The FEniCS project version 1.5, *Archive of numerical software* 3 (100) (2015).
- [5] C. Geuzaine, J. Remacle, Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities, *International Journal for Numerical Methods in Engineering* 79 (11) (2009) 1309–1331.
- [6] S. Jiménez-Alfaro, E. Martínez-Pañeda, A computational framework for predicting the effect of surface roughness in fatigue, *International Journal of Fatigue* 199 (2025) 109044.