AN INVERSE PROBLEM FOR DAMAGED BRITTLE MATERIALS

Nicola Ferro

Politecnico di Milano
September 2015

Contents

1 Introduction

The study of the propagation of cracks in brittle materials has been object of massive interest lately. Both Mechanics and Mathematics found challenging questions to answer, from a practical and theoretical point of view.
Historically, the question has been posed in these terms: how does a pre-existing crack propagate in a brittle material under stress? The answer has got obvious implications, especially in applied engineering. Nevertheless the importance of the mathematical models is quite as strong.
Variational models have been suffering from the critical description of the crack path since the first formulation. The kick-off on the subject is due to Griffith, who proposed a two-term energy for the material: an elastic part and an energy cost related to the crack propagation. The result is in the form: $\int_{\Omega\setminus\Gamma} \Psi(\epsilon) + \int_{\Gamma} \Phi$, where Γ identifies the path of the crack and Ψ and Φ are appropriate functions.
Francfort and Marigo followed his footsteps until Ambrosio and Tortorelli furnished one of the most used model. This one is reliable, well-posed and, above all, does not require any predictions on the crack path. The winning key of this model - that pleased engineers and theorists - is that it allows us to make the crack "field" smooth and easier to model. In practice, the sharp crack is substituted by a smooth phase field.
The basic, essential part of this approach is the Γ-convergence of the smooth functional to Francfort and Marigo's one.
The aim of this work is to implement a scheme for the resolution of an inverse problem in a context like this: linear elasticy in which a damage field evolves. Indeed many studies are present in the literature about elastic continua, but not much about the model we will present can be found.

In this report we will firstly introduce the mathematical setting leading to the final model. Thus we will obtain the direct problem which will be our starting point.

In the following part, the inverse problem is briefly derived using some usual techniques.

In the third part the resolutive algorithm is presented and we will explain how the code should work focusing on some critical aspects in the implementation. We will also give a closer look at the structures and solver used in the program.

In the forth part the complete Python code will be commented along with a brief tutorial containing instructions about the parameters to be changed to test it. At the end of this section we will show some results obtained on benchmark cases and will list some *caveats* concerning the physical parameters involved and how they should (or should not) be changed.

Since all the code is written in Python, we want to somehow adhere more strictly to the Course Program. For this reason, in the Appendix we will schematically and quickly compare the implementation of the program in Python with its counterpart in C++. Pros and cons will be weighted both from a didactic and practical point of view.

2 The mathematical model

Our focus is on the shrinking of brittle materials due to a thermal shock.
Following what done in [1, 2], we use a gradient damage model in which the damage field is represented by the scalar variable α. In particular, this continuous variable has the following meaning: $\alpha = 0$ identifies sound material, whereas $\alpha = 1$ stands for fully damaged material.
With this description it is easy to understand that cracks are located where the damage variable varies roughly from 0 to 1 and back to 0 in a small part of the domain.
The variational problem follows from the minimization of the energy functional which takes into account elastic behaviour together with thermal properties. The functional reads as follow:

$$\mathcal{J}(\mathbf{u}, \alpha) = \frac{1}{2}\int_\Omega (1-\alpha)^2 A(\varepsilon(\mathbf{u}) - \varepsilon_{th}) : (\varepsilon(\mathbf{u}) - \varepsilon_{th}) + \int_\Omega \frac{G_c}{4c_w}\left(\frac{\alpha}{l} + l|\nabla\alpha|^2\right) \tag{1}$$

In (1):

- $\Omega \subset \mathbb{R}^2$ is the domain, $\mathbf{u} \in \mathbb{R}^2$ represents the displacement and $\alpha \in \mathbb{R}$ is the damage variable as described above;
- $(1-\alpha)^2$ identifies the fracture model chosen by the author. It is a simple but satisfactory one;
- A is the isotropic stiffness tensor of the 4^{th} order: $A(\cdot) = \lambda tr(\cdot)\mathbf{I} + 2\mu(\cdot)$, with λ, μ Lamé's constants;
- $\varepsilon(\mathbf{u}) = \dfrac{\nabla\mathbf{u} + \nabla\mathbf{u}^T}{2}$ is the linear strain tensor.
- $\varepsilon_{th} = \beta(T_t - T_0)\mathbf{I}$ is the inelastic deformation due to the thermal shock. It is clearly time-dependent, as T_t is the temperature field at time t.
- G_c, c_w and l are the parameters of the model. In particular, l is the internal length which can be used as the reference unit.

The first integral represents the elastic energy of the material, while the second one is the energy associated to the cracks.
T_t is the explicit time-dependent part of the problem and it is the solution of the parabolic heat equation

$$\begin{cases} \dfrac{\partial T_t}{\partial t} - k_c \Delta T_t = 0 & \text{on } \Omega \\ T_t = T_0 - \Delta T & \text{on } \partial\Omega \\ T(\mathbf{x}, 0) = T_0 & \text{on } \Omega \end{cases} \tag{2}$$

We should notice how the damage field is not considered when solving the equation for the temperature. This means that the generation and the propagation of the cracks do not influence the thermal behaviour of the material. Clearly, no phase changes are considered in the model.

Finally the problem, for every discrete timestep t_i, is

$$\min_{\mathbf{u}, \alpha_i \geq \alpha_{i-i}} \mathcal{J}_{t_i}(\mathbf{u}, \alpha) \tag{3}$$

The constraint on α is used in order to match the physical irreversibility of cracks, as observed experimentally: the damage field can only grow as no remargination is allowed.
It is correct to underline that the model proposed is one of a kind: no initiation of the domain for the cracks to propagate is needed, while every other model require initial flaws in the material.
As far as the optimization problem is concerned, we look for local minima of the energy functional for two main reasons: on one hand it is prohibitively expensive to look for global minima, on the other it is more meaningful to take into account all minimizers.
However we have to underline that Γ-convergence ensures the convergence only to global minima.

Proposition 1. *The functional is Fréchet-differentiable in $[H^1(\Omega)]^2 \times (H^1(\Omega) \cap L^\infty(\Omega))$. The derivative along the directions $(\boldsymbol{v}, \beta)$ is the following:*

$$\mathcal{J}'(\boldsymbol{u}, \alpha; \boldsymbol{v}, \beta) =$$
$$= \underbrace{\int_\Omega (1-\alpha)^2 A(\varepsilon(\boldsymbol{u}) - \varepsilon_{th}) : \varepsilon(\boldsymbol{v})}_{a(\alpha; \boldsymbol{u}, \boldsymbol{v})} +$$
$$+ \underbrace{\int_\Omega \left((\alpha - 1)\beta A(\varepsilon(\boldsymbol{u}) - \varepsilon_{th}) : (\varepsilon(\boldsymbol{u}) - \varepsilon_{th}) + \frac{c\beta}{l} + 2cl \nabla\alpha \cdot \nabla\beta \right)}_{b(\boldsymbol{u}; \alpha, \beta)}$$

where $c := \dfrac{G_c}{4c_w}$.

Definition 1. *The pair $(\boldsymbol{u}, \alpha)$ is a critical point for the functional $\mathcal{J}$ if $\mathcal{J}'(\boldsymbol{u}, \alpha; \boldsymbol{v}, \beta) = a(\alpha; \boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{u}; \alpha, \beta) = 0 \quad \forall (\boldsymbol{v}, \beta) \in [H^1(\Omega)]^2 \times (H^1(\Omega) \cap L^\infty(\Omega))$.*

3 The inverse problem

Deriving from an applied engineering field, elasticity problems often benefit from the existence (and abundance) of real, measured data along with simulated ones. Thanks to this, it is legitimate to ask how numerical simulations could match physical results in the best possible way.
The search for some parameters of the phenomenon at hand is a common practice and the study of inverse problems is, of course, the most important, mathematically rigorous technique to achieve the results we seek.
The problem we presented in the previous section depends on many constants and most of them are strictly related to the material we are considering. Some others are modellistic parameters and are chosen in order to make the model accurate enough.
In this fashion, we can consider l a double-faced parameter: it is clearly a parameter that defines how good the approximation takes place (i.e. $l \to 0^+$ in the Γ-convergence procedure), but, at the same time, it may be associated to a particular material [2].
To each value of internal length l a material can be linked. The search for an esteem of a parameter like this is not standard in linear elasticity or, at least, is not very common since physical parameters are always preferred and - to be completely honest - sometimes make more sense.
On the other hand, the problem is very sensitive (as shown in the pictures below) with respect to the choice of the internal length and both qualitative and quantitative differences can be observed with a different choice of l.
For these reasons, our focus will be on the formulation of an inverse problem to estimate the internal length.

3.1 The formulation

For the derivation of the inverse problem, we use a classical approach. It is based on the Lagrangian of the objective function which we want to minimize. Thus, it is clear that the starting point is the proper choice of the objective function. We usually tend to choose to minimize the difference between measured and simulated data which are mostly affected by the change of the parametes we want to estimate.
Of course this is not an easy task and different choices may be equally acceptable. In our case, a change in the internal length leads to a change in the number of fractures which develop inside the body. Since it is not simple to carry on a calculation like this, we chose to analyze the following objective function:

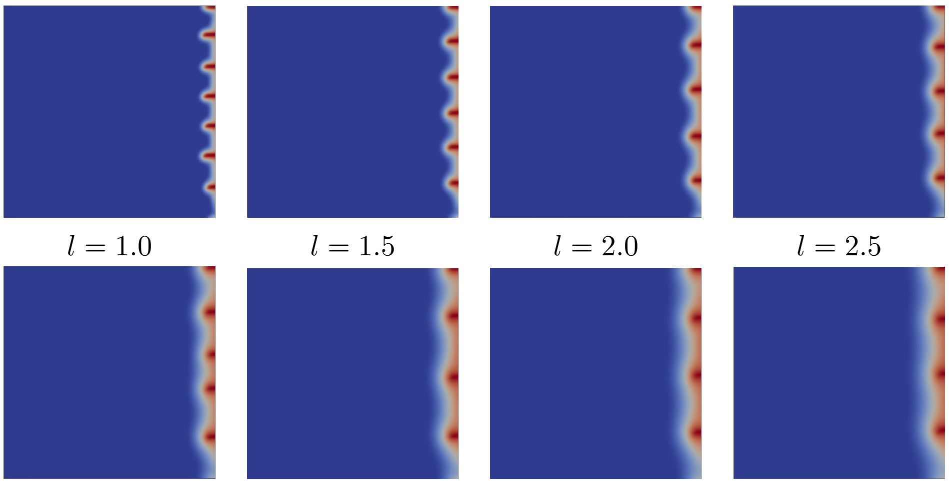

Table 1: Sensitivity of the problem with respect to the internal length.

$$\mathcal{T}(\alpha) = \frac{1}{2|\Omega|}\left(\int_\Omega (\alpha_M - \alpha)\right)^2$$

where α_M is the measured damage field.
We now add the constraints by using Lagrange multipliers and the state equations. The resulting functional depends on three variables which will be considered independent for the derivation of three problems.

$$\mathcal{L}((\mathbf{u},\alpha),(\mathbf{v},\beta),l) = \frac{1}{2|\Omega|}\left(\int_\Omega (\alpha_M - \alpha)\right)^2 + \int_\Omega (1-\alpha)^2 A(\varepsilon(\mathbf{u}) - \varepsilon_{th}) : \varepsilon(\mathbf{v})$$
$$+ \int_\Omega \left((\alpha - 1)\beta A(\varepsilon(\mathbf{u}) - \varepsilon_{th}) : (\varepsilon(\mathbf{u}) - \varepsilon_{th}) + \frac{c\beta}{l} + 2cl\nabla\alpha\cdot\nabla\beta\right),$$

where $(\mathbf{v},\beta)$ represents the couple of Lagrangian multipliers.
By studying the stationarity of $\mathcal{L}$, we obtain the variational problems we look for.

Definition 2. *Fréchet - differentiating the Lagrangian with respect to its indipendent variables, the following equations are obtained:*

$$\begin{cases} \dfrac{\partial\mathcal{L}}{\partial(\boldsymbol{v},\beta)}\cdot\delta(\boldsymbol{v},\beta) = 0 & \forall\delta(\boldsymbol{v},\beta)\in\mathcal{U}\times\mathcal{D} \quad \textit{State problem} \\ \dfrac{\partial\mathcal{L}}{\partial(\boldsymbol{u},\alpha)}\cdot\delta(\boldsymbol{u},\alpha) = 0 & \forall\delta(\boldsymbol{u},\alpha)\in\mathcal{U}\times\mathcal{D} \quad \textit{Adjoint problem} \\ \dfrac{\partial\mathcal{L}}{\partial l}\cdot\delta l = 0 & \forall\delta l : \Omega\to\mathbb{R}^+ \quad \textit{Control problem} \end{cases}$$

In particular,
State problem:
Find $(\boldsymbol{u},\alpha) \in \mathcal{U} \times \mathcal{D}$ *such that*

$$\begin{cases} \displaystyle\int_\Omega (1-\alpha)^2 A(\varepsilon(\boldsymbol{u})-\varepsilon_{th}) : \varepsilon(\delta\boldsymbol{v}) = 0 & \forall \delta\boldsymbol{v} \in \mathcal{U} \\ \displaystyle\int_\Omega \Big((\alpha-1)\delta\beta A(\varepsilon(\boldsymbol{u})-\varepsilon_{th}) : (\varepsilon(\boldsymbol{u})-\varepsilon_{th}) + \frac{c\delta\beta}{l} + 2cl\nabla\alpha\cdot\nabla\delta\beta \Big) = 0 & \forall \delta\beta \in \mathcal{D} \end{cases}$$

Adjoint problem:
Find $(\boldsymbol{v},\beta) \in \mathcal{U} \times \mathcal{D}$ *such that*

$$\begin{cases} \displaystyle\int_\Omega (1-\alpha)^2 A\varepsilon(\boldsymbol{v}) : \varepsilon(\delta\boldsymbol{u}) + \int_\Omega 2(\alpha-1)\beta A(\varepsilon(\boldsymbol{u})-\varepsilon_{th}) : \varepsilon(\delta\boldsymbol{u}) = 0 & \forall \delta\boldsymbol{u} \in \mathcal{U} \\ |\Omega|^{-1} \int_\Omega (\alpha_M - \alpha) \displaystyle\int_\Omega \delta\alpha + \int_\Omega 2(\alpha-1)A(\varepsilon(\boldsymbol{u})-\varepsilon_{th}) : \varepsilon(\boldsymbol{v})\delta\alpha & \\ + \displaystyle\int_\Omega \beta A(\varepsilon(\boldsymbol{u})-\varepsilon_{th}) : (\varepsilon(\boldsymbol{u})-\varepsilon_{th})\delta\alpha + \int_\Omega cl\nabla\beta\cdot\nabla\delta\alpha = 0 & \forall \delta\alpha \in \mathcal{D} \end{cases}$$

Control problem:
Find $l : \Omega \to \mathbb{R}^+$ *such that*

$$\int_\Omega -\frac{c\beta}{l^2}\delta l + \int_\Omega c\nabla\alpha\cdot\nabla\beta\delta l = 0 \quad \forall \delta l : \Omega \to \mathbb{R}^+.$$

in which $\mathcal{U}$ *and* $\mathcal{D}$ *are functional spaces defined as follows:*

$$\mathcal{U} = \{\boldsymbol{u} \in [H^1(\Omega)]^2 \textit{ such that } \boldsymbol{u} = \boldsymbol{g} \textit{ on } \partial\Omega_D\},$$
$$\mathcal{D} = \{\alpha \in H^1(\Omega) \textit{ such that } \alpha = \gamma \textit{ on } \partial\Omega_D, 0 \le \alpha \le 1\}.$$

The control problem may be written in a strong form taking into account the generality of the test function. It reads:

$$-\frac{c\beta}{l^2} + c\nabla\alpha\cdot\nabla\beta = 0 \qquad \text{in } \Omega$$

or, in a more useful way,

$$l^2 = \frac{\beta}{\nabla\alpha\cdot\nabla\beta} \qquad \text{in } \Omega$$

provided that $\nabla\alpha\cdot\nabla\beta \neq 0 \quad \forall \mathbf{x} \in \Omega$.

3.2 The time dependence

So far we have considered the problem in static using the hypothesis of *quasistaticity.* While this may seem correct because no explicit time dependence is involved and every single timestep must be solved indipendently, we must

consider a proper definition of the set $\mathcal{D}$.
As we have already explained in section 2, a condition of physical irreversibility must be respected. That is, each damage field depends on the one computed at the previous timestep. In a continuous setting:

$$1 \geq \alpha(t) \geq \alpha(\bar{t}) \geq 0 \quad \forall \bar{t} \leq t.$$

Thus, the set $\mathcal{D}$ is precisely $\mathcal{D}(t)$ in which the lower and upper bound for the functions in it are now $\alpha(\bar{t}), \forall \bar{t} \leq t$ and 1 respectively.

Remark 1. *Both in the state problem and in the adjoint problem the time dependence is hidden in* ε_{th}.

It is now clear that we cannot escape from dealing with time dependencies. Moreover, we must underline that the most significant features are displayed in the solution field only after a certain t^*.
t^* is the instant when the bifurcation of the unstable damage solution takes place and cracks begin to appear and evolve.
Thus, if we want to capture the phenomenon in its peculiar characteristics, we need to consider an observation time large enough for the field to be completely developed.
To summarize, both the state and the adjoint problems are constrained and the constraints are time-dependent. This remark will be essential in the numerical implementation.

4 The algorithm

4.1 The workflow

The algorithm we used is classical and follows strictly the formulation of the previous section. The solution scheme is:

$$\text{State problem } \rightarrow \text{Adjoint problem } \rightarrow \text{Control problem}$$

Thus, every problem can benefit from the information derived from the solution of the previous problem in the chain. In particular, the control problem is easily solved thanks to the strong form of the equation.
The main issues of the algorithm are the time dependence, as seen in section 3, and the need of a solver which deals with lower and upper bounds.
We now present the flow of the pseudocode to enlighten some crucial aspects.

```
while $\eta >$ TOLL do
    $t = dt$;
    while $t \leq t_f$ do

        SOLVE DIRECT PROBLEM for $(\mathbf{u}, \alpha)$;

        SOLVE ADJOINT PROBLEM for $(\mathbf{v}, \beta)$;

        SOLVE CONTROL PROBLEM for $\tilde{l}$;

        $t = t + dt$;

    end while

    $l_{new} = l - \theta \tilde{l}$;
    $\eta = \frac{|l - l_{new}|}{l}$;

end while
```

Many things look still unclear:

- how do we solve the direct and the adjoint variational problems? That is, how do we deal with constraints?
- How do we recover a space-independent $\tilde{l}$ from space-dependent variables in the control problem?
- How do we get a time-independent $\tilde{l}$ from time-depedent variables?
- How do we choose the step factor θ?

Let us give a closer look at the solvers and how they are used inside the algorithm.
First of all we need to underline that all the variational problems involved, i.e. *primal* and *dual*, are solved by classical **finite element methods**. FEniCs, interfaced with Python, is a very useful tool which we will exploit for the discretization of the problems.
In addition, Python and FEniCs provide quite easy tools to handle simple constraints, thanks to useful interfaces with PETSc libraries. In this way, solving for the primal α and for the dual β is essentially an easy task. However, one should not forget that the two problems are systems of two equations each and only one per system is constrained.
For this reason, we adopt a fixed-point-like scheme, as descripted below:

```
STATE PROBLEM:
for each timestep,
  while $\tau > \mathtt{TOLL}_\alpha$ do
      SOLVE VARIATIONAL PROBLEM for $\mathbf{u}$;
      SOLVE CONSTRAINED PROBLEM for $\alpha$;
      $\tau = ||\alpha - \alpha_{old}||_{L^\infty(\Omega)}$;
  end while
```

The same procedure may be used to solve the adjoint problem:

```
ADJOINT PROBLEM:
for each timestep,
  while $\sigma > \mathtt{TOLL}_\beta$ do
      SOLVE VARIATIONAL PROBLEM for $\mathbf{v}$;
      SOLVE CONSTRAINED PROBLEM for $\beta$;
      $\sigma = ||\beta - \beta old||_{L^\infty(\Omega)}$;
  end while
```

Remark 2. *Each iterative scheme presented above has got an extra check in the while loop. In order to be sure that the loop finishes, we impose a maximum number of iteration, as commonly done in the literature.*

Now solving the control problem for $\tilde{l}$ is as easy as it seems. What we get from this problem, for each timestep, is a function $\tilde{l} : \Omega \to \mathbb{R}^+$. We need to obtain a value $\tilde{l} \in \mathbb{R}^+$ without any space-dependence.
We chose to average the function over the entire domain:

$$\tilde{l} = \frac{1}{|\Omega|} \int_\Omega \tilde{l}(\mathbf{x}) d\mathbf{x}.$$

Other solutions can be considered: for instance the domain of integration could be reduced to the only part of Ω really interested by the phenomenon. However, even with this expedient, we obtain a discrete sequence $\tilde{l}(t)$ since both primal and dual solutions depend on time. This is not what we are

looking for because it is an implicit hypothesis that physical and modellistic parameters do not change in our timescale.
We can tackle the problem in different ways. Two of them are the most significant [4]:

- $\tilde{l} := \tilde{l}(t_f)$;
- $\tilde{l} := \langle \tilde{l}(t) \rangle$;

where $\langle \cdot \rangle$ is a temporal mean operator.
As we have already explained, the final instants are the most significant ones since the phenomenon is well displayed and developed. Thus, we decided to use an hybrid solution for the calculation of $\tilde{l}$:

$$\tilde{l} = \langle \tilde{l}(t) \rangle_{t_f - kdt}^{t_f} := \frac{1}{kdt} \int_{t_f - kdt}^{t_f} \tilde{l}(\xi) d\xi.$$

This choice is also due to our will to exclude the first steps of the simulation which are - of course - important but do not show marked features because the bifurcation has not taken place yet.
A choice like this leads to the use of only the last k solutions and to discard the previous ones. We will try to avoid all this useless expense: we will go into more detail in the next paragraph.

Remark 3. *We will use the easiest and straightforward formula for time integration and the formula above will reduce to:*

$$\tilde{l} = \frac{1}{k} \sum_{\xi = t_f - kdt}^{t_f} \tilde{l}(\xi) d\xi.$$

As far as the step θ is concerned, we need to admit that the choice is problem-dependent: it depends on the magnitude of $\tilde{l}$ and l.
A first criterium is simply based on the observation that $l_n^2 = l_{n-1}^2 - \theta \tilde{l}$ and that $l_n \in \mathbb{R}^+$.
Another suggestion could be that the smaller n is the greater θ can be picked.
It is clear that, in this case, proceeding by heuristic is the safest way, but a deeper study could help find an optimal steplength that ensures convergence and speed.
In our case, a bit of experience and some sensitivity analysis of the problem led to this choice:

$$\theta = \frac{l_{n-1}^2 - 1}{10^{\lfloor \log_{10}(\tilde{l}) \rfloor + 1}}$$

4.2 Some remarks

As said before, we want to exploit the last time iterations of primal and dual solutions. Thus, it seems quite useless to solve both the problems forward in time. So, we decided to solve the adjoint problem backward in time for just k iterations as they are the only ones we need to perform the time average. In this way, we save computational time and memory resources.

As far the steplength is concerned, we need to make some points:

- after a fixed number of iterations, θ is lowered in order to slow down the procedure which could be too *aggressive*;
- due to the constraints on β, $\tilde{l}$ is (almost) always positive, so we cannot estimate the internal length from initial guesses which are lower than the target. However, the study of the derivative of the objective function alone can help find the right direction for the estimate;
- an inner check for the derivative of the objective function is performed to avoid huge gradients near the solution that can ruin the process.

5 The code

5.1 A quick step-by-step explanation

In this section we will analyze some extracts from the complete code and we will comment the most unusual things in the implementation.
First of all, we get an overview of the *modules* we need to make the code work.
What is a module? A module is a python file (extension .py) that contains definitions, declarations, functions or classes. They are imported in a file using the import keyword.
Along with some built-in modules, we need:

- direct.py;
- inverse.py;
- data.py;
- Borders.py;
- MyProblems.py;
- Solvers.py;
- energies.py
- utilities.py;

direct.py is the main program for the solution of the direct problem. It is needed in order to obtain the simulated data to match. Precisely, since we do not have real data, we firstly simulate them and consider them as measured.
inverse.py is the main program for the solution of the inverse problem.
data.py contains the declarations of some constants needed in different modules. In this way, it is possible to change some parameters directly in this file, affecting all the simulations. It is like the variables here defined are global (or you can see it as an easy GetPot file which stores all the initial definitions) and seen by the main programs.
Borders.py contains four useful classes which define the border of a rectangular domain. They are derived from the abstract class SubDomain.
MyProblems.py is a module with two classes for the optimization problem. They are derived from some abstract classes and are specilized for the case at hand.
Solvers.py contains three classes to solve the heat equation, the primal and the dual problem. They all consist of a constructor to set the variational forms and of a method solve to invoke the solutor.
In **energies.py** the elastic energy and its derivative are computed inside some functions.
Finally, **utilities.py** is a file with useful functions used throughout the codes.

Remark 4. *We have divided solvers and problems in two different files because each one groups a specific type of class. MyProblems.py classes are derived ones whereas Solvers.py classes are specific classes completely written from scratch.*
Indeed in Python it is possible to use a single file to group many different entities and classes, but, in our opinion, this division looks clearer and easier to access.

We now list some extracts from the codes to comment them. The documentation of all the functions and classes can be found in the folder Documentation after typing in the command line:

```
doxygen config.dox
```

Borders.py

```
class Left(SubDomain):

        def __init__(self, LL, HH):
                self.L = LL
                self.H = HH
                SubDomain.__init__(self)

        def inside(self, x, on_boundary):
                return near(x[0], 0.)
```

We derived some classes from the SubDomain base class (in the extract only the Left class is shown). It consists only of a constructor and a method to verify whether we are on the border or we are not.

Solvers.py

```
aT =
T*w*dx + dt*inner(kc*0.5*grad(T), grad(w))*dx
self.LT =
(T_old)*w*dx - dt*inner(kc*0.5*grad(T_old), grad(w))*dx
```

The variational formulation is easily written in Python using Dolfin forms. We use a Crank-Nicholson's scheme for the time discretization. Since the matrix of the discretized PDE does not change with time, it is assembled outside the temporal loop, while the right hand side will be assembled just before the solutor is invoked at each time step.

energies.py

```
def E_duM
        (uM, vM, duM, alphaM, Tsol,
        mu, Lambda, l, Gc, c_w, beta, T0):

        eth = beta*(Tsol-T0)*Identity(2)

        elastic_energyM =
        0.5*inner(sigma(eps(uM),alphaM, mu, Lambda) -
        sigma(eth,alphaM, mu, Lambda), eps(uM)-eth)*dx

        dissipated_energyM =
        Gc/(4.*c_w)*(alphaM/l +
        l*dot(grad(alphaM), grad(alphaM)))*dx

        total_energyM =
        elastic_energyM + dissipated_energyM

        E_uM = derivative(total_energyM,uM,vM)
        return replace(E_uM,{uM:duM})
```

We exploit a practical tool by Dolfin: it is possible to declare a functional and let Dolfin do the Fréchet derivative for you. Our case is very simple, but it can be somehow useful at certain points.

direct.py/inverse.py

```
from dolfin import *
import [...]

if not has_petsc_tao():
```

```
print("DOLFIN must be compiled with TAO.")
exit(0)
```

These preliminary instructions in the main programs are needed in order to include the *modules* we will use throughout the code. Since we will use PETSc TAOSolver, we check whether Dolfin is compiled with TAO. Without TAO the optimization problems cannot be initialized, so the program aborts with exit value 0.

direct.py/inverse.py

```
V_u = VectorFunctionSpace(Th, "CG", 1, ndim)
V_alpha = FunctionSpace(Th, "CG", 1)
```

The declaration of the finite element spaces is very trivial. We use $\mathbb{P}^1$-type elements both for the displacement and the damage field (and the other scalar variables).

direct.py/inverse.py

```
f = open('results/alphaM%d.pckl'%(timestep), 'w')
pickle.dump(alphaM.vector().array(), f)
f.close()
```

We use the built-in module **pickle** to store the data in an easy format Python can deal with.

inverse.py

```
os.system("python direct.py %f" %l)
```

We invoke this command to run direct.py from inverse.py with l as command line parameter. We made this choice to completely separate the first simulation which stores the measured data from the proper inversion algorithm.

utilities.py

```
def steplength(step, ltilde, iterinv):

        P = 1./250
                if step > 0:
                        P =
                        (ltilde**2-1.)/
                        10**(int(math.log10(step))+1)
                if iterinv >=5:
                        P /= 4.
                if iterinv >=25:
                        P /= 10.

                return P
```

Finally, for the sake of completeness, this is the function for the calculation of the steplength for each iteration of the inverse problem.

5.2 A brief tutorial

Once Python (version 2.7.6) and Dolfin (version 1.5.0) are installed, launching the program is as easy as it seems:

```
python inverse.py 1 10
```

The code has few requirements in input: in particular, the user should provide a target internal length (to perform the direct simulation since we do not have real data), say l, and the initial guess l_0 since the procedure is iterative. Both should be positive (and the range [1, 8] is the most significant). Other parameters (as those in **data.py**) should be accessible to the user, such as the size of the mesh or the elastic constants, but the choice of parameters such these is very delicate. The combination of "right" parameters can let the direct problem work, while other choices bring to the simulation of a wrong phenomenon. In particular, the accessible and modifiable parameters are only the internal length and the initial guess. Indeed this may seem quite strict and makes the program rigid. Of course this argument looks right, but we can try to justify this choice.
The main problem of modifying the physical setting of the problem is the model itself. Only specific combinations of the parameters are responsible for the success of the direct simulation. The formation and evolution of the cracks in the material depend on a bifurcation process due to instability of the elastic solution. For this reason, not every choice works well. Moreover, the size of the mesh must be fixed such that the phenomenon is well detected. That is, the mesh has to be fine with respect to the internal length (mesh size $\simeq l/4$ is ideal). Thus, our suggestion is to keep the physical parameters and vary l in the range [1, 8] unless you have a deep insight of the model. In this last case, simple changes in the Griffth energy density or in other constants result in (hopefully!) correct simulations.

For the sake of completeness, we want to underline that the simulations with larger values of l are faster than those with smaller l. In addition, we need to analyze the impact of the time dependence of the simulation and the backward resolution of the adjoint problem. Indeed the need of storing the data for the target simulation and for the tentative damage field for every timestep is basic and result in large time simulation and memory usage. At the same time this is unavoidable as the complete developed field is more significant than the first instants.

Remark 5 (**Extensions**). *The definition of the total energy of the body in **energies.py** makes it easy to change the damage model, even if the one we use looks quite reasonable. In addition, the code can be quickly extended to the 3D case. In this case simulations are very expensive and parallelization could be the winning key to solve the issue.*

5.3 Numerical results

We tested the code different times with different input parameters. The results are all reasonable and satisfying.
Here we list some numerical tests we performed:

```
python inverse.py 2 4

Number of iterations: 11
Estimate:             2.00971
% Error:              0.5 %
```

```
python inverse.py 1 0.5

Number of iterations: 26
Estimate:             1.07394
% Error:              7 %
```

```
python inverse.py 3 1

Number of iterations: 48
Estimate:             3.01295
% Error:              0.4 %
```

```
python inverse.py 4 6

Number of iterations: 36
Estimate:             4.01308
% Error:              0.3 %
```

```
python inverse.py 4 9

Number of iterations: 50
Estimate:             4.014458
% Error:              0.3 %
```

```
python inverse.py 2.5 5

Number of iterations: 10
Estimate:             2.48908
% Error:              0.5 %
```

A APPENDIX

In this appendix we want to explore the main differences between an implementation of such a program using FEniCs with Python or with C++11. As implied in the introduction, there are both didactic differences along with practical ones.

Using the C++ interface or the Python interface is quite similar, but the first one gives a deeper insight of what is happening inside the code and which data structures are being used. Python, instead, is indeed more a black box than C++, to be completely honest. On the other hand, Python is more direct and helps avoid some tricky stuff which should be tackled in a C++ program. We can now list some differences between the codes:

Python

```python
class BetaProblem(NonlinearProblem):

        def __init__(self, FB, JB, BETA):
                NonlinearProblem.__init__(self)
                self.FBeta = FB
                self.JBeta = JB
                self.Beta = BETA

        def F(self, b, x):
                self.Beta.vector()[:] = x
                assemble(self.FBeta, tensor=b)

        def J(self, A, x):
                self.Beta.vector()[:] = x
                assemble(self.JBeta, tensor=A)
```

C++

```cpp
class OptProb : public NonlinearProblem
{
        public:

        OptProb(const Form& f, const Form& j):
                L(f), a(j) {}

        void F(GenericVector& b, const GenericVector& x)
        {
                assemble(b, L);
        }

        void J(GenericMatrix& A, const GenericVector& x)
        {
                assemble(A, a);
        }

        private:

        const Form& L;
        const Form& a;

};
```

The difference is only in the syntax as the two classes have a constructor and some assembler calls.
The striking difference is in the public/private keyword. Why does not Python need them? The web has its philosophical answer:

It's cultural. In Python, you don't write to other classes' instance or class variables. In Java (or C++), nothing prevents you from doing the same if you really want to - after all, you can always edit the source of the class itself to achieve the same effect. Python drops that pretence of security and encourages programmers to be responsible. In practice, this works very nicely.

user Kirk Strauser on stackoverflow.com

The C++ class is derived from an abstract class whose definition can be found on GitHub.

Python

```
import pickle

f = open('results/alpha%d.pckl'%(timestep), 'w')
pickle.dump(alpha.vector().array(), f)
f.close()
```

C++

```
File f('results/alpha%d.pvd'%(timestep), 'w');
f << alpha;
```

Once again, here the difference is not in the way command are used, but in the format we use to store the information.

Whereas Python can handle .pvd file as well (as it is one of the most used format), using pickle is very handy but not universal.

Python

```
solver_alpha.solve
(AlphaProblem(), alpha.vector(),
lb_alpha.vector(), ub_alpha.vector())
```

C++

```
std::size_t solve(OptimisationProblem& problem,
        PETScVector& x, const PETScVector& lb,
        const PETScVector& ub);
```

The last comparison is just a consideration derived from the snippets above: which structures Python uses? Which is the type of your variables? You may not know that and your code still works.
Indeed C++ definitions and type policy make the user (and the coder!) more aware of what she is using.

References

1. Bourdin, B., Marigo, J.-J., Maurini, C., Sicsic, P., *Morphogenesis and Propagation of Complex Cracks Induced by Thermal Shocks*, Physical Review Letters, 2014.

2. Marigo, J.-J., Maurini, C., Sicsic, P., *Initiation of a periodic array of cracks in the thermal shock problem: A gradient damage modeling*, Journal of the Mechanics and Physics of Solids, 2013.

3. Langtangen, H., P., *A FEniCs Tutorial*, FEniCs project, 2011.

4. Na, S.-W., Kallivokas, L., F., *Direct time-domain soil profile reconstruction for one-dimensional semi-infinite domains*, Soil Dynamics and Earthquake Engineering, 2009.

5. Bonnet., M., Constantinescu, A., *Inverse problems in elasticity*, Institute of Physics Publishing, Inverse Problems, 2005.