POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Solving Nonlinear Elliptic Equations with `femR`

Advanced Programming for Scientific Computing

Authors: **Carlotta Gatti - Emanuele Tamburini**

Advisor: Professor Laura Maria Sangalli
Co-advisors: Professor Eleonora Arnone
Aldo Clemente
Alessandro Palummo
Academic Year: 2022-23

Contents

Introduction

The aim of our project is to enlarge the capabilities of the `R`/`C++` library `fdaPDE` and the `R`/`C++` library `femR`, in order to solve Advection Diffusion Nonlinear-Reaction Partial Differential Equations. In particular, we want to extend the library `femR` enabling `R` users to solve these kind of equations through a user-friendly interface.

The `fdaPDE` library, in its `core` module, is already able to deal with Elliptic Partial differential equations of the kind

$$-\text{div}(Ku) + \mathbf{b} \cdot \nabla u + cu = f$$

where u is the function we want to solve for, f is a forcing term and K, $\mathbf{b}$ and c are some parameters describing the physical problem underneath the PDE. Such equations are then embedded and exploited in more complex statistical models which are constructed in the higher modules of the library `fdaPDE`. These other modules, however, make large use of the code in the `core` section.

Although Elliptic PDEs can already model complex physical problems from the world applications, we are interested in expanding the capabilities of the library in order to solve equations that contain a nonlinear reaction term. Being able to solve such equations, would indeed extensively broaden the class of statistical models that can be used to tackle problems on both 2D and 3D domains. Namely, we are considering Partial Differential Equations of the form:

$$-\text{div}(Ku) + \mathbf{b} \cdot \nabla u + h(x, u)u = f \tag{1}$$

where now the function h represents the nonlinearity in the equation.

In particular, throughout the project, we are going to focus on a particular variation of the well known *Fisher-KPP equation* [23] which is stationary and presents a forcing term

$$-\nu\Delta u + \alpha(1 - u)u = f$$

where ν and $\alpha \in \mathbb{R}$ are parameters.

Our project is structured as follows:

- *Chapter 1* Nonlinear-Reaction Partial Differential Equations: we recall the theoretical background to deal with Elliptic Nonlinear-Reaction Partial Differential Equations from a numerical point of view and we prove the well posedness of the considered mathematical problem.
- *Chapter 2* Numerical Methods for Nonlinear Equations: we review what methods in the literature can be exploited to manage complex nonlinear equations under different aspects.

In these first two chapters, we present the notation that will be used in the following chapters as well.

- *Chapter 3* Nonlinear Methods for PDEs: in this chapter, we analyse how the numerical nonlinear methods from the literature can be applied to the specific PDE 1.2 with two main different strategies.
- *Chapter 4* Code Structure: we present our code implemented in the library as well as the main ideas that drove our implementation choices.
- *Chapters 5- 6* Numerical Results *and* Conclusions: in these chapters we show our results with numerical experiments and draw conclusions on them.
- *Chapter 7* Installation of `femR` and Instructions: this chapter provides useful instructions to the interested reader who wants to run the code themselves.

1 | Nonlinear-Reaction Partial Differential Equations

As previously introduced, the goal of this project is to be able to solve Diffusion Nonlinear-Reaction Partial Differential Equations of the form

$$-\text{div}(Kf) + \mathbf{b} \cdot \nabla f + h(x, f)f = u(x) \tag{1.1}$$

with the `femR` library. In particular, in this project, we focus on the following specific problem

$$\begin{cases} -\nu\Delta u + \alpha(1-u)u = f & \Omega \\ u = g & \partial\Omega \end{cases} \tag{1.2}$$

for which we prove the well posedness in section 1.1. Problem 1.2 is a particular variation of the well known *Fisher-KPP equation* [23] which is stationary and presents a forcing term.

Models based on the Fisher's equation can be found in diverse scientific fields, such as Population Dynamics, Genetics and Biochemistry [12], [23]. In particular, one specific application for modelling the dynamics of the $\tau-$protein in the human brain can be found in [3]. Other applications, in which nonlinear reaction terms different from 1.2, are recurrent in literature, and one other example can be found in [21]. Our work, upon proof of well posedness for the mathematical problem, can be addressed for any kind of Nonlinear-Reaction equation of the form 1.1. In this chapter we prove well posedndess for our problem in section 1.1 while in section 1.2 we recall some theoretical result of the Galerkin Finite Elements method.

1.1. Well Posedness

We are concerned in proving the existence of a solution for the problem

$$\begin{cases} -\nu\Delta u + \alpha(1-u)u = f & \Omega \\ u = 0 & \partial\Omega \end{cases} \tag{1.3}$$

where $\Omega \subset \mathbb{R}^n$ is a bounded, Lipschitz domain ($n = 2, 3$), ν,$\alpha \in \mathbb{R}_+$. Notice that we are dealing with homogeneous Dirichlet boundary conditions. The case of non-homogeneous Dirichlet boundary conditions can be reduced to this case through a lifting procedure.

A weak solution for problem 1.3 is a function $u \in H_0^1(\Omega)$ such that

$$\nu \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} + \alpha \int_\Omega (1-u)uv \, d\mathbf{x} = \int_\Omega fv \, d\mathbf{x} \quad \forall\, v \in H_0^1. \tag{1.4}$$

The proof for the existence we propose for problem 1.4 relies on the well known theorem by Schauder [18], that we report here for convenience.

Theorem 1.1. *(Schauder): Let X be a Banach space. We are given:*

- *$A \subset X$ compact and convex,*
- *$T : A \to A$ continuous operator.*

Then T has a fixed point $x \in A$.

Theorem 1.2. *(Existence): We claim that there exists at least one solution $u \in H_0^1$ for problem 1.4 whenever $\Omega \subset \mathbb{R}^n$ and $n \leq 3$.*

Proof. The ideas and procedures exploited for this proof follow [17], theorem 9.14, and [18], theorem 10.10. To start with, we define the space $V \equiv H_0^1$ and rewrite problem 1.4 in the following weak formulation:

$$\text{find } u \in V \text{ s.t. } a(u,v) - g(u,u,v) = F(v) \quad \forall v \in V \tag{1.5}$$

where

- $a(u,v) = \nu \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x} + \alpha \int_\Omega uv \, d\mathbf{x}$,
- $F(v) = \int_\Omega fv \, d\mathbf{x}$,
- $g(w,u,v) := \alpha \int_\Omega wuv \, d\mathbf{x}$.

The first step consists in linearizing problem 1.5 as follows:

$$\text{given } w \in V, \text{ find } u \in V \text{ s.t. } a(u,v) = F(v) + g(w,w,v) \quad \forall v \in V, \tag{1.6}$$

for which we can state that

- $a(u,v)$ is a continuous and bounded bilinear form (if $\alpha > 0$),
- $F(v)$ is a bounded linear functional,
- $g(w,w,v)$ is continuous:
  using the generalized Hölder inequality with three terms

$$\left| \int_\Omega fgh \right| \leq \|f\|_p \|g\|_q \|h\|_r \quad \text{with } p,r,q > 1 \text{ and } \frac{1}{p} + \frac{1}{q} + \frac{1}{r} = 1, \tag{1.7}$$

  choosing $p = 4$, $q = 4$, $r = 2$, we obtain

$$|g(w,w,v)| = \alpha \left| \int_\Omega w^2 v \right| \leq \alpha \|w\|_4^2 \|v\|_2$$

  and from the Sobolev Embedding theorem we know that $V \hookrightarrow L^4$ with compact embedding, thus

$$\|\cdot\|_4 \leq \tilde{c}_4 \|\cdot\|_V. \tag{1.8}$$

  Finally, it is easy to show that

$$|g(w,w,v)| \leq \alpha \|w^2\|_V^2 \|v\|_V.$$

Therefore, we can then use the Lax-Milgram theorem to prove that, for every $w \in V$, the linearized problem 1.6 admits a unique solution

$$u = T(w)$$

where $T : V \to V$ is a linear operator. Moreover,

$$\|u\|_V = \|T(w)\|_V \leq \frac{1}{\nu} \left(C_p \|f\|_2 + \alpha \|w\|_2^2 \right) =: M.$$

Now, we are left with a fixed point problem. Indeed, given $w \in V$, the function $u = T(w)$, solution of 1.6, will be a solution of 1.5 if and only if it is a fixed point for the map $T : V \to V$, i.e. $u = T(u)$. The next steps consist in showing that T admits a fixed point.

We define the set $B_M = \{\xi \in V : \|\xi\|_V \leq M\}$, which is a closed, bounded, convex subset of V. In order to apply the Schauder theorem 1.1 to $T : B_M \to B_M$, we must check that the latter is compact and continuous.

To show that T is **compact**, let $\{w_k\}_{k=0}^{\infty}$ be a bounded sequence in B_M, i.e. $\|w_k\|_V \leq M \,\forall k$. We want to prove that there exists a converging subsequence $\{T(w_{k_j})\}_{j=0}^{\infty}$ in B_M.

Let w_1, $w_2 \in B_M$ and, consequently, $u_1 = T(w_1)$, $u_2 = T(w_2)$; we can write

$$\nu \int_\Omega \nabla \left[T(w_1) - T(w_2)\right] \cdot \nabla v + \alpha \int_\Omega \left[T(w_1) - T(w_2)\right] v = g(w_1, w_1, v) - g(w_2, w_2, v),$$

$$\nu \int_\Omega \nabla \left[T(w_1) - T(w_2)\right] \cdot \nabla v + \alpha \int_\Omega \left[T(w_1) - T(w_2)\right] v = \alpha \int_\Omega w_1^2 v - \alpha \int_\Omega w_2^2 v = \alpha \int_\Omega (w_1^2 - w_2^2) v.$$

Choosing the test function $v = T(w_1) - T(w_2) \in V$, we get

$$\nu \int_\Omega \nabla \left[T(w_1) - T(w_2)\right]^2 + \alpha \int_\Omega \left[T(w_1) - T(w_2)\right]^2 = \alpha \int_\Omega (w_1 - w_2)(w_1 + w_2)(T(w_1) - T(w_2)).$$

Exploiting again the generalized Hölder inequality 1.7 on the right hand side, with $p = 4$, $q = 2$ and $r = 4$, we obtain

$$\min(\nu, \alpha)\|T(w_1) - T(w_2)\|_V^2 \leq \alpha \|w_1 - w_2\|_4 \|w_1 + w_2\|_2 \|T(w_1) - T(w_2)\|_4.$$

Finally, using the Sobolev Embedding inequality 1.8 and recalling that $\|w\|_2 \leq \|w\|_V \leq M$, we obtain

$$\min(\nu, \alpha)\|T(w_1) - T(w_2)\|_V^2 \leq 2\alpha M \tilde{c}_4 \|w_1 - w_2\|_4 \|T(w_1) - T(w_2)\|_V,$$

therefore

$$\|T(w_1) - T(w_2)\|_V \leq \frac{2\alpha M \tilde{c}_4}{\min(\nu, \alpha)} \|w_1 - w_2\|_4. \tag{1.9}$$

Thanks to the compactness of the embedding $V \hookrightarrow L^4$, there exists a converging subsequence $\{w_{k_j}\}_{j=0}^{\infty}$ in L^4

$$\|w_{k_{j_1}} - w_{k_{j_2}}\|_4 \to 0 \quad \text{as} \quad j_1, j_2 \to \infty,$$

which means that, from 1.9, the sequence $\{T(w_{k_j})\}$ is a Cauchy sequence, hence convergent in V.

The **continuity** of the operator T can be proved using once again the Sobolev Embedding

inequality 1.8 on 1.9

$$\|T(w_1) - T(w_2)\|_V \leq \frac{2\alpha M \tilde{c}_4^2}{\min(\nu, \alpha)} \|w_1 - w_2\|_V,$$

from which we obtain that T is Lipschitz continuous with constant $\frac{2\alpha M \tilde{c}_4^2}{\min(\nu,\alpha)}$ and this completes the proof. □

1.2. Galerkin Finite Element Approximation

For this project, we have been working with Galerkin Finite Elements to discretize and solve the Diffusion Nonlinear-Reaction Equation 1.2. In this section, we shall briefly recall the theoretical framework of the Galerkin Finite Element method, that we will leverage in the following chapters.

As previously mentioned, let $V \equiv H_0^1(\Omega)$, with $\Omega \subset \mathbb{R}^n$, $n = 2, 3$. The weak form 1.4 can be rewritten as

$$\text{find } u \in V \text{ s.t. } a(u, v) + b(u, u, v) = F(v) \quad \forall\, v \in V,$$

where

- $a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, d\mathbf{x}$,
- $b(u, w, v) = \int_\Omega h(u) w v \, d\mathbf{x}$,
- $F(v) = \int_\Omega f v \, d\mathbf{x}$.

To discretize the problem with the Galerkin method, we introduce a finite dimensional subspace $V_\mathtt{h}$, described by a parameter $\mathtt{h}$, such that

$$V_\mathtt{h} \subset V \quad \dim(V_\mathtt{h}) = N_\mathtt{h} < \infty \quad \forall\, \mathtt{h} > 0.$$

In particular, we define the subspace $V_\mathtt{h}$ to be the space generated by the set of basis functions $\{\varphi_i\}_{i=1}^{N_\mathtt{h}}$, i.e. $V_\mathtt{h} = \text{span}\{\varphi_1, \dots \varphi_{N_\mathtt{h}}\}$. The discretized problem on the finite dimensional subspaces, can be written as follows

$$\text{find } u_\mathtt{h} \in V_\mathtt{h} \text{ s.t. } a(u_\mathtt{h}, v_\mathtt{h}) + b(u_\mathtt{h}, u_\mathtt{h}, v_\mathtt{h}) = F(v_\mathtt{h}) \qquad \forall\, v_\mathtt{h} \in V_\mathtt{h}. \tag{1.10}$$

Actually, since every $v_\mathtt{h} \in V_\mathtt{h}$ can be written as a linear combination of the basis functions

of $V_{\mathtt{h}}$, it suffices that the equation above is satisfied for every φ_i:

$$\text{find } u_{\mathtt{h}} \in V_{\mathtt{h}} \text{ s.t. } a(u_{\mathtt{h}}, \varphi_i) + b(u_{\mathtt{h}}, u_{\mathtt{h}}, \varphi_i) = F(\varphi_i) \qquad \forall\, i = 1, \dots, N_{\mathtt{h}}.$$

Moreover, we can write $u_{\mathtt{h}}$ as a linear combination of the basis functions too:

$$u_{\mathtt{h}} = \sum_{j=1}^{N_{\mathtt{h}}} \varphi_j U_j$$

for some unknown coefficients $U_j \in \mathbb{R}$, which yields us to the following algebraic formulation

$$\sum_{j=0}^{N_{\mathtt{h}}-1} a(\varphi_j, \varphi_i) U_j + \sum_{j=0}^{N_{\mathtt{h}}-1} b\left(\sum_{\ell=0}^{N_{\mathtt{h}}-1} U_\ell \varphi_\ell, \varphi_j, \varphi_i\right) U_j = F(\varphi_i) \quad \forall\, i = 1, \dots, N_{\mathtt{h}}.$$

The latter can be interpreted as a nonlinear system of equations

$$[A + H(\mathbf{U})]\mathbf{U} = \mathbf{F} \tag{1.11}$$

where:

- $\mathbf{U} = [U_1, \dots, U_{N_{\mathtt{h}}}]^T$,
- $\mathbf{F} = [F_1, \dots, F_{N_{\mathtt{h}}}]^T$ with $F_i = \int_\Omega f\varphi_i \, d\mathbf{x}$,
- $A_{ij} = \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \, d\mathbf{x}$,
- $H(\mathbf{U})_{ij} = \int_\Omega h\left(\sum_\ell U_\ell \varphi_\ell\right) \varphi_i \varphi_j \, d\mathbf{x}$.

The Finite Elements method consists in a particular choice of the subspaces for the test functions. In particular, given a triangulation of Finite Elements $\mathcal{T} = \{K_j\}_{j=0}^{\mathtt{ndofs}}$ of the domain Ω, where `ndofs` represents the number of degrees of freedom, we can introduce the family of Finite Elements subspaces

$$X_{\mathtt{h}}^r = \{v_{\mathtt{h}} \in \mathcal{C}^0(\bar{\Omega}) : v_{h|K_j} \in \mathbb{P}^r \quad \forall\, K_j \in \mathcal{T}\},$$

where r defines the order of the Finite Elements.

Finally, if we define the space $V_{\mathtt{h}} = \{v_{\mathtt{h}} \in X_{\mathtt{h}}^r : v_{\mathtt{h}} = 0 \text{ on } \partial\Omega\}$, and we choose φ_i to be the lagrangian basis functions, the formulation 1.10 becomes the Finite Elements formulation of problem 1.2.

2 | Numerical Methods for Nonlinear Equations

In this chapter, we want to explore known methods in the literature to cope with nonlinear equations or systems of nonlinear equations. In particular, we are interested in finding the root(s) for the most general nonlinear problem

$$\text{find } \tilde{x} \in \mathbb{R}^n \text{ s.t. } F(\tilde{x}) = 0$$

where $F : \mathbb{R}^n \to \mathbb{R}^n$. We will mention unconstrained minimisation problems as well, since the two concepts are strictly related and can be used interchangeably in specific scenarios. Minimisation problems can be written as follows:

$$\text{find } \tilde{x} \in \mathbb{R}^n \text{ s.t. } \tilde{x} = \min_{x \in \mathbb{R}^n} f(x)$$

where $f : \mathbb{R}^n \to \mathbb{R}$. This chapter has three sections. Section 2.1 reviews the Newton method for solving nonlinear equations. In Section 2.2 we introduce the quasi-Newton methods that we use later to solve a Nonlinear-Reaction PDE. Section 2.3 discusses two strategies that we use for making the quasi-Newton methods globally convergent, meaning that we have the guarantee of local convergence to a local optimum for every choice of the initial data.

2.1. Newton Methods

Let us consider the problem

$$\text{given } F : \mathbb{R}^n \to \mathbb{R}^n, \text{ find } \tilde{x} \in \mathbb{R}^n \text{ s.t. } F(\tilde{x}) = 0. \tag{2.1}$$

The Newton's method is an iterative method for which, at each iteration k, we compute the solution of an affine approximation of F. From the fundamental theorem of calculus,

we know that

$$F(x+s) = F(x) + \int_0^1 J(x+ts)s\,dt$$

where $J(x)$ represents the jacobian matrix of the function F. We build an approximate model of the function F by approximating the integral with $J(x)s$.

Therefore, at each iteration, we solve the linear model given by

$$M_k(s) = M(x_k + s) = F(x_k) + J(x_k)s,$$

which can also be seen as a first order Taylor approximation of the function F. Let $N \geq 0$ be the iteration index, then the Newton direction δ_N can be computed by solving $M_k(s^N) = 0$, which leads to the solution of the linear system

$$J(x_k)s^N = -F(x_k). \tag{2.2}$$

Once the Newton direction is obtained, the iterative method will update the solution at the next step as

$$x_{k+1} = x_k + s^N. \tag{2.3}$$

The Newton Method can also be applied to unconstrained minimisation problems. In this framework, the problem is formulated as

$$\text{given } f : \mathbb{R}^n \to \mathbb{R}, \text{ find } \tilde{x} \in \mathbb{R}^n \text{ s.t. } \tilde{x} = \min_{x \in \mathbb{R}^n} f(x) \tag{2.4}$$

In this case, the Newton method is applied to a quadratic approximation of f. Specifically, we rely on a second order Taylor approximation of the function f

$$m_k(s) = f(x_k) + (\nabla f(x_k))^T s + \frac{1}{2} s^T H(x_k) s. \tag{2.5}$$

The matrix $H(x)$ in equation 2.5 represents the hessian matrix of the function f. The Newton direction is given by the minimum of 2.5. Since the hessian matrix is positive definite, this corresponds to the solution of the linear system $H(x_k)s^N = -\nabla f(x_k)$. The current iterate is finally updated as in 2.3.

As it is well known in the literature, the Newton method exhibits local convergent properties. Moreover, $J(x_k)$ may not be available in practice and an estimate of it may be mandatory for some applications. In the next sections, we will review some strategies to overcome these two issues.

2.2. Quasi-Newton Methods

With quasi-Newton methods, we handle situations where the Jacobian matrix is not available in its exact form. This happens when the function F is complex, not provided in an analytic form, or when computing the derivatives using finite differences would be too computationally expensive.

Among various quasi-Newton methods, we specifically consider the secant method as our starting point. This one-dimensional quasi-Newton approach exhibits a competitive superlinear order of convergence, with an order $q = \varphi = \frac{1+\sqrt{5}}{2}$. To present the approach, consider the one-dimensional problem

$$\text{given } f : \mathbb{R} \to \mathbb{R}, \text{ find } \tilde{x} \in \mathbb{R} \text{ s.t. } f(\tilde{x}) = 0.$$

we employ the Newton method as

$$x_{k+1} = x_k - [f'(x_k)]^{-1} f(x_k)$$

and the secant method approximates $f'(x_k)$ using the secant equation:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}. \tag{2.6}$$

In the subsequent section 2.2.1, we investigate the generalization of this method to multiple dimensions.

2.2.1. Broyden Method

Broyden Method is the most successful secant-method extension to solve systems of nonlinear equations [4]. Addressing again the problem 2.1, what needs to be approximated at every iteration is the Jacobian matrix J. We will call the approximation of such matrix the Broyden Matrix:

$$B_k \approx J(x_k).$$

The secant equation 2.6, in the multi-dimensional case, translates in the following linear system:

$$B_k \ (x_{k+1} - x_k) = F(x_{k+1}) - F(x_k), \tag{2.7}$$

in which the matrix is the unknown. Therefore, if $n > 1$, there are infinitely many matrices satisfying the secant equation, living in a subspace of dimension $n(n-1)$, and the secant

equation does not completely specify how we should choose B_k. A simple workaround for this inconvenient could be to write the secant equation for the $n-1$ points at the previous iterations. However, the Broyden method avoids this expensive procedure by leveraging on the idea that, at every iteration, the only information we have about the Jacobian matrix comes from its approximation at the previous iteration. So, we should choose the matrix B_{k+1} among all those that are feasible, in order to minimise the quantity

$$\|B_{k+1} - B_k\|.$$

In 1965, C.G.Broyden proposed the so-called *Broyden update formula*:

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)s_k^T}{s_k^T s_k} \tag{2.8}$$

where $s_k = x_{k+1} - x_k$ and $y_k = F(x_{k+1}) - F(x_k)$. Broyden update is the minimum change to B_k consistent with the secant equation 2.7. In particular, equation 2.8 solves the problem

$$B_{k+1} = \min_{B\,:\,Bs_k = y_k} \|B - B_k\|$$

when $\|\cdot\|$ is the ℓ_2 matrix norm and, moreover, it solves the problem uniquely if $\|\cdot\|$ is the Frobenius norm [4].

The Broyden method gives a recipe to find B_{k+1} at every iteration. However, we should mention that it is possible to avoid the solution of the linear system 2.2 at every iteration, by taking advantage of the Sherman-Morrison formula, which gives us the inverse of a *rank-1 update* to a matrix.

Proposition 2.1. *Given a squared matrix $A \in \mathbb{R}^{n\times n}$ and a pair of column vectors $u, v \in \mathbb{R}^n$, then $A + uv^T$ is invertible $\iff 1 + v^T A^{-1} u \neq 0$ and the inverse is given by:*

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1} u}. \tag{2.9}$$

The Broyden update can indeed be characterized as a *rank-1 update* to the matrix B_k and therefore we can apply formula 2.9 on the matrix B_{k+1}. Finally, the inverse of the Jacobian can be obtained at each iteration as:

$$B_{k+1}^{-1} = \prod_{j=0}^{k} \left(I + \frac{s_{j+1} s_j^T}{s_j^T s_j} \right)$$

where $s_j = x_{k+1} - x_k$.

The Broyden method discussed so far falls under the name of *good Broyden method*, as it first approximates the Jacobian and then compute its inverse. There exists also a version of the Broyden method called *bad Broyden method* which approximates directly the inverse of the Jacobian matrix with the update

$$B_{k+1}^{-1} = B_k^{-1} + \frac{s_k - B_k^{-1} y_k}{y_k^T y_k} y_k^T$$

which minimizes the Frobenius norm $\|B_{k+1}^{-1} - B_k^{-1}\|$. This latter version of the Broyden method can lead to better performance in some applications [10].

2.2.2. Modified Broyden-like Method

In this section, we review the so called *modified Newton methods*, and in particular *modified quasi-Newton methods*, as they represent a powerful tool for the solution of nonlinear system of equations. Modified Newton algorithms are iterative methods of the form

$$x_{k+1} = x_k + s^N + s^{MN}$$

where s^N is the Newton direction described in this chapter, and s^{MN} is the modified Newton direction given by:

$$s^{MN} = J_F(x_k)^{-1} F(z_k), \quad \text{where} \quad z_k = x_k + s^N.$$

Modified Newton methods are known to achieve cubic convergence rate if the Jacobian matrix $J_F(\tilde{x})$ is Lipschitz-continuous and nonsingular at the solution $\tilde{x}$. Recently, in [**?**], the idea of modified Newton methods has been tested on quasi-Newton methods as well, with satisfactory results. We focus our attention on the modified Broyden-like quasi-Newton method, which makes use of the modified Broyden-like formula given by

$$B_{k+1} = B_k + \theta_k \frac{(y_k - B_k s_k) s_k^T}{s_k^T s_k} \tag{2.10}$$

where $s_k = x_{k+1} - x_k$ and $y_k = F_{k+1} - F_k$ as in the classic Broyden formula 2.8, and the parameter θ_k is chosen such that $|\theta_k - 1| \leq \theta$ for some $\theta \in (0, 1)$. It has been shown [11] that with this formula, the sequence of matrices B_{k+1} converges to the exact Hessian matrix at the solution $\tilde{x}$, which is not guaranteed for classic Broyden methods.

We report here the pseudo algorithm of the modified Broyden-like method that we took as reference for the `C++` implementation. This algorithm introduces various parameters,

whose values have to be fine-tuned manually. In our `C++` implementation we considered the work done in [26] as a reference for such values.

Algorithm 2.1 Modified Broyden-like Method

choose the sequence $\{\varepsilon\}_{k=0}^{\infty}$ s.t. $\sum_{k=0}^{\infty}\varepsilon_k < \varepsilon < \infty$
choose $x_0 \in \mathbb{R}^n$, $B_0 \in \mathbb{R}^{n\times n}$ non singular, $\sigma_1 > 0$, $\sigma_2 > 0$, $\tau > 0$, $r \in (0,1)$, $\theta \in (0,1)$, $\rho \in (0,1)$
while $(\|F_k\| \neq 0)$ **do**
  compute Broyden direction: $s^B = -B_k^{-1}F_k$
  compute $z_k = x_k + s^B$
  if $(\|F(z_k)\| > \tau\|F_k\|)$ **then**
    use broyden direction: $s_k = s^B$
  else
    compute modified Broyden direction: $s^{MB} = -B_k^{-1}F(z_k)$
    if $\left(\|F(z_k + s^{MB})\| \leq \rho\|F_k\| - \sigma_1\|s^B + s^{MB}\|^2\right)$ **then**
      use modified Broyden direction: $s_k = s^B + s^{MB}$
      set $\alpha_k = 1$
      set `skiplinesearch = True`
    else
      use Broyden direction $s_k = s^B$
    end if
  end if
  if $(\|F(z_k + s_k)\| \leq \rho\|F_k\| - \sigma_1\|s_k\|^2)$ **and not** `skiplinesearch` **then**
    $\alpha_k = 1$
  else if not `skiplinesearch` **then**
    $\alpha_k = \max\{1, r, r^2, \ldots\}$ with $\|F(x_k + \alpha s_k)\| \leq \|F_k\| - \sigma_2\|\alpha s_k\|^2 + \varepsilon_k\|F_k\|$
  end if
  $x_{k+1} = x_k + \alpha_k s_k$
  update B_{k+1} as in 2.10; `k++;`
end while

2.3. Global Strategies for quasi-Newton Methods

While quasi-Newton methods solved the problem of the unavailability of the Jacobian matrix in the Newton method, in this section we will address the second issue of the Newton method: the local convergence property. As it is well known, the Newton Method doesn't give any guarantee of convergence if the initial guess is not *sufficiently* close to the

solution of the problem. Some strategies have been developed to fix the local convergence property and we have analyzed two of the most famous approaches. However, we must warn the reader that, even if such strategies aim at giving global convergence properties to Newton Methods, it is not guaranteed that the algorithms will always lead to a solution. In general, establishing how many roots a nonlinear function has and compute those roots, is a complex task. Given a certain method, it is always possible to find a counter example function for which the algorithms fails, given finite computational resources.

For the sake of clarity of exposition, while addressing the global strategies in the following subsections, we will focus on an unconstrained minimization problem represented by equation 2.4. In particular, we focus on the following simple test case nonlinear function, proposed in [4] and revisited in our project:

$$f : \mathbb{R}^2 \to \mathbb{R},\ f(x_1, x_2) = (x_1^2 + x_2^2 - 2)^2 + (e^{x_1 - 1} + x_2^3 - 2)^2 \tag{2.11}$$

which has a minimum in $x = [1, 1]^T$. Global strategies will be adapted for nonlinear equations of the form 2.1 with the merit function transformation $f = \frac{1}{2}\|F\|_2^2$.

Before starting with the analysis, we recall the definition of descent direction that will be used in the next subsections: a direction s is said to be a *descent direction* if $f(x_k + s) < f(x_k)$. This characterization is equivalently expressed by the negativity of the directional derivative: $\nabla f(x_k)^T s < 0$. Finally, it's worth noting that an effective global strategy should minimise the number of evaluations of the function f. In fact, while computing the value of the objective function f at multiple points provides extremely useful information, it is also considered costly and should be avoided when unnecessary.

2.3.1. Line Search

The most simple and natural approach to achieve global convergence with Newton Methods is probably the Line Search strategy.

With the Newton method, at every iteration, we compute the descent direction minimizing the quadratic model associated with the nonlinear function at point x_k. While we are sure that the Newton direction is a descent direction for the model function 2.5, we have no guarantee that this direction will be a descent direction for the original nonlinear function as well. In practice, most of the times it will not, leading to a divergent behaviour of the Newton method. The Line Search idea is that of modifying the length of the step in the Newton direction, so that it is effectively a descent direction for the original nonlinear

function. The update 2.3 becomes:

$$x_{k+1} = x_k + \alpha_k s^N$$

where a_k is the parameter provided by the Line Search algorithm. One simple approach to determine the parameter α_k is to require that the reduced step is a descent direction:

$$f(x_k + \alpha_k s^N) < f(x_k).$$

However, it can be shown with a counter example that this condition is not sufficient to achieve convergence, and we need more sophisticated constraints to be impose on α, which are the so called *Wolfe Conditions*. In particular, the first condition 2.12 (also called itself *Armijo condition*), is a condition of sufficient decrease, while 2.13 is the curvature condition and represents a bottom limit for α_k.

$$f(x_k + \alpha_k s_k) \leq f(x_k) + c_1 \alpha_k [\nabla f(x_k)]^T s_k \tag{2.12}$$

$$\nabla [f(x_k + \alpha_k s_k)]^T s_k \geq c_2 [\nabla f(x_k)]^T s_k \tag{2.13}$$

with $0 < c_1 < c_2 < 1$.

In practical codes, we only need the condition 2.12, if we determine α_k with the *Backtracking* approach. With the Backtracking Line Search, we first start with $\alpha_k = 1$ and decrease it multiplying by a parameter $\rho \in (0, 1)$ until the step is compliant with the Wolfe condition 2.12. A pseudo algorithm for a Backtracking Line search strategy is provided:

Algorithm 2.2 Backtracking Line Search

1: Choose $\rho \in (0, 1)$, $c \in (0, 1)$, set $\alpha = 1$
2: **while** $f(x_k + \alpha s_k) \leq f(x_k) + c\alpha [\nabla f(x_k)]^T s_k$ **do**
3: $\quad \alpha = \rho \alpha$
4: **end while**
5: return α

Finally, the figure 2.1 gives a graphical idea of how a Line Search algorithm would work on the toy example 2.11. In this example, we can see that the Newton direction is the best minimizer for the quadratic function, but not for the nonlinear function f. A Line Search strategy will find α_k such that a shorter step in the Newton direction is performed at iteration k.

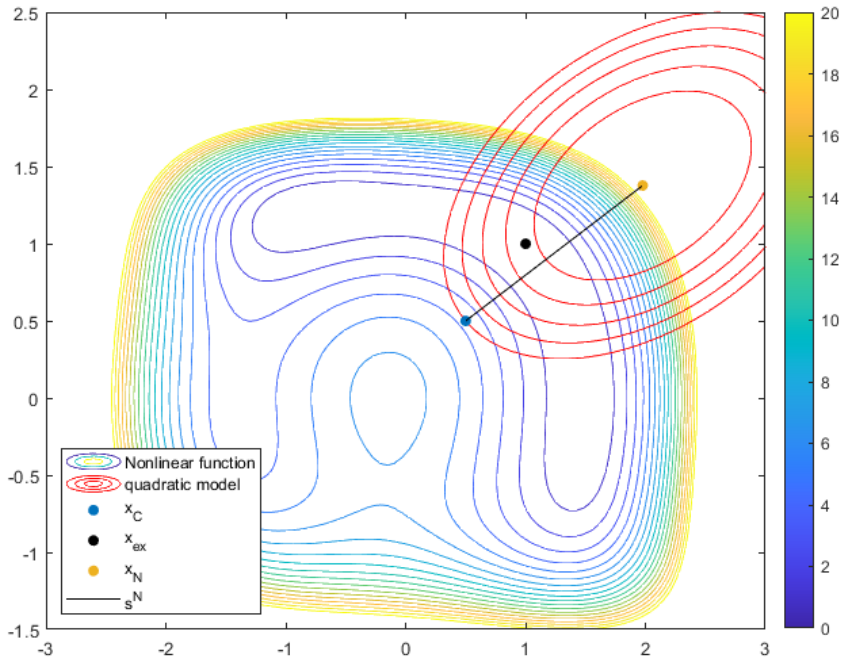

Figure 2.1: Example of Line Search approach to minimise a nonlinear function $f : \mathbb{R}^2 \to \mathbb{R}$

2.3.2. Trust Region

Another method that has been widely developed in the literature of global strategies, is the Trust Region method. With this approach, we introduce a dynamic parameter Δ_k that represents the radius of a ball centered in x_k. This neighbourhood of point x_k represents the region in which we trust the quadratic model 2.5 to be a good approximation of the nonlinear function f. The main difference with respect to Line Search methods, is that now we minimise the quadratic model using the Trust Region as a constraint. So, instead of first minimising the quadratic model, and then focusing on the Newton direction to find α_k, we first restrict the quadratic model, and then minimise it inside the Trust Region. Be aware that the direction found in this case is, in general, different from the Newton direction. Therefore, with Trust Region methods, not only we shorten the Newton step to satisfy the Wolfe Conditions 2.12, 2.13, but also we steer it towards the point that minimises the nonlinear function the most, changing its direction. Figure 2.2 gives a graphical representation of this concept for the toy example 2.11.

For Trust Region methods, an informed choice of the parameter Δ_k is crucial for the algorithm to work as expected. To this purpose we introduce a useful indicator for the quadratic model:

$$\rho_k = \frac{f(x_k) - f(x_k + s_k)}{m_k(0) - m_k(s_k)} \tag{2.14}$$

which is the ratio between the *actual reduction* (of the function f) and the *predicted reduction* (of the model m_k). If this ratio is close to 1, it means that the model is a good approximation for our function, therefore we should expand our region of trust. Similarly, if ρ_k is far from 1, we should restrict the trust region, as the model is predicting a reduction that is not happening in practice. The challenge that arises at each iteration with Trust Region methods consists in minimising the quadratic model with the constraint of the Trust Region, namely:

$$\min_{s\in\mathbb{R}^n} m_k(s) = \min_{s\in\mathbb{R}^n} f(x_k) + [\nabla f(x_k)]^T s + \frac{1}{2} s^T H(x_k) s \quad \text{s.t.} \quad \|s\| \leq \Delta_k. \tag{2.15}$$

Clearly, solving this sub problem exactly at each iteration would be too expensive, considering that the primary goal is to solve the minimisation problem 2.4. To overcome this issue, we can use some numerical methods to solve 2.15 approximately. The most diffused approach to confront the problem 2.15 at each iteration, and the one we implemented in our code, is the so-called *dogleg method*, which can be used whenever the Hessian matrix is symmetric and positive definite. There exist several variations of this method, but we will report here the one we reviewed.

The first step of the *dogleg method* is that of computing the Cauchy Point (or Cauchy direction). To do so, we solve a linear approximation of the subproblem 2.15:

$$s^* = \min_{s\in\mathbb{R}^n} f(x_k) + [\nabla f(x_k)]^T s \quad \text{s.t.} \quad \|s\| \leq \Delta_k.$$

Multiplying the result by a factor τ_k ensures that $m_k(\tau_k s^*)$ is minimised within the Trust Region bound. Practically, s^* can be computed in closed form as

$$s^* = -\frac{\Delta_k}{\|\nabla f_k\|} \nabla f_k.$$

Whenever the Hessian matrix H of the function f is symmetric and positive definite, the parameter τ_k is computed as

$$\tau_k = \min \left\{ \frac{\|\nabla f_k\|^3}{\Delta_k [\nabla f_k]^T H_k \nabla f_k}, 1 \right\}.$$

There exist other methods to find τ_k when the Hessian matrix is not symmetric and positive definite. However, note that, as we anticipated before, in our project we are concerned in solving a nonlinear system of equations of the form 2.1 and, when we tackle this problem as a minimisation task, we define the Hessian matrix as $H_k = J_F(x_k)^T J_F(x_k)$. This means that, for the problem we are working on, the Hessian matrix is always symmetric

and positive definite by construction. Finally, the Cauchy Point will be determined by

$$s^C = \tau_k s^*. \tag{2.16}$$

Obviously, a linear approximation of the problem 2.15 is not satisfactory to our purposes. After all, the Cauchy direction represents a particular choice of the Steepest Descent step length, which will give us a linear convergence rate. If we want to achieve superlinear convergence rates, we need to exploit the Hessian matrix, as it contains precious curvature information of the function f. The *dogleg method* combines both the Newton (s^N) and Cauchy (s^C) directions to solve the problem 2.15, and its step is given by

$$s^D(\xi) = \begin{cases} \xi s^C & 0 \leq \xi < 1 \\ s^C + (\xi - 1)(s^N - s^U) & 1 \leq \xi \leq 2 \end{cases} \tag{2.17}$$

where ξ is another parameter that ensures the step length respects the Trust Region constraint. The latter can be computed by solving the quadratic problem

$$\|s^C + (\xi - 1)(s^N - s^C)\|^2 = \Delta_k^2. \tag{2.18}$$

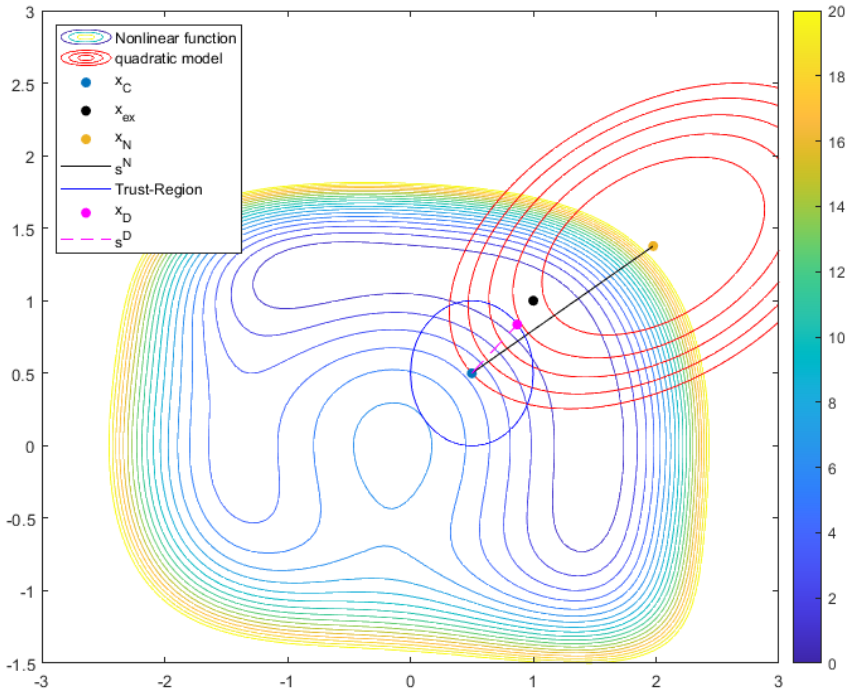

Figure 2.2: Trust Region approach to minimise a nonlinear function $f : \mathbb{R}^2 \to \mathbb{R}$

At last, we report here the pseudo algorithm that we take as reference for our `C++` implementation.

Algorithm 2.3 Trust-Region

initialise Broyden method
for $k = 1, 2, \ldots$ **do**
  compute Newton direction s^N with Broyden method
  compute Cauchy direction s^C as in 2.16
  while dogleg step not accepted **do**
    if s^N is inside Trust Region **then**
      use Newton direction: $s_k = s^N$
    else
      solve quadratic problem 2.18
      compute dogleg direction: $s_k = s^D$ as in 2.17
    end if
    compute ρ_k as in 2.14
    if $\rho_k << 1$ **or** $\rho_k >> 1$ **then**
      restrict Trust-Region
    else if $|\rho_k - 1| \approx 0$ **then**
      expand Trust-Region
    end if
    if $f_{k+1} > f_k$ **then**
      reject step and retry
    else
      accept dogleg step: $x_{k+1} = x_k + s_k$
    end if
  end while
  finalise Broyden method, `k++`;
end for

3 | Nonlinear Methods for PDEs

The nonlinear numerical methods that we have discussed so far, have a straightforward application on nonlinear equations and systems of nonlinear equations. However, how to take advantage of these methods while working with Partial Differential Equations is not so obvious, as there are several approaches, all with their advantages and disadvantages. In this chapter, we want to investigate two approaches for dealing with Nonlinear-Reaction PDEs and analyze their strengths and weaknesses under several points of view. Following the purpose of our project, we will focus on the Diffusion Nonlinear-Reaction equation of interest, which is equation 1.2. In section 3.1 we investigate how to solve the nonlinearity at the PDE level, while in section 3.2 we discuss how more general Nonlinear-Reaction PDEs can be solved, dealing with the nonlinearity at the algebraic level.

3.1. Nonlinearity at the Strong Level

Probably, the most straightforward family of methods for solving a nonlinear PDE consists in solving the nonlinearity in the strong form of the equation, before discretizing it with a Galerkin method. Let us consider the problem 1.2; in this case, we know that the nonlinearity is represented only by the reaction term and, in particular, by the function $h : V \to V$. We can get rid of the nonlinearity at the strong level by building a sequence of linearized PDEs that will converge to the original nonlinear one. The simplest method that we can employ to build such sequence, is the *Fixedpoint Method*, also known as *Picard Iteration Method*. As it is very often done in the literature, following the idea of the fixedpoint method for nonlinear equations, we simply use a known term to solve the nonlinearity h, and therefore linearize the PDE as follows:

$$-\nu\Delta u_{k+1} + h(u_k)u_{k+1} = f. \tag{3.1}$$

Starting from an initial guess u_0, we solve this sequence of linear PDEs indexed by k and, at every step, we inform the nonlinearity of the next step with the current solution. As it is well known [17], the Fixedpoint method presents a convergent behaviour if and only

if the operator $T : u_k \mapsto u_{k+1}$ admits a fixed point. Moreover it is a global method and the solution is unique if the operator T is also a contraction. This result is well known in the literature as *Banach Fixedpoint Theorem* [6], that we report here for convenience.

Theorem 3.1. *Let X be a Banach space and $\mathcal{T} : X \to X$ a γ-contraction, then:*

- *there exists a unique $\bar{x} \in X$ such that $\mathcal{T}(\bar{x}) = \bar{x}$,*
- *for every $x_0 \in X$, the sequence $x^{(k+1)} = \mathcal{T}(x^{(k)})$ converges to $\bar{x}$ as*

$$\|x_k - \bar{x}\|_X \leq \gamma^k \|x_0 - \bar{x}\|_X$$

In this latter case, there is no constraint on the initial guess u_0. As we showed in chapter 1, this is the case for the particular choice of the nonlinearity we are considering, $h(u)u = (1-u)u$. Note that the uniqueness of the solution can be asserted whenever the Lipschitz constant in 1.9, which depends on the data f, is less or equal than 1.

Although simple, the fixedpoint method guarantees at most a linear order of convergence. For this reason, we proceed to exploit once again the Newton method to get rid of the nonlinearity at the strong level of the PDE, this method can be applied to the strong formulation of the PDE in order to build a sequence of linearized PDEs, converging to the original nonlinear equation. The idea is to exploit the notion of differentiability in Banach spaces, which was introduced in the early 20th century by Fréchet and further developed by Gâteaux [15],[1],[24]. We recall the definition of Gâteaux Derivative employed in the application of the Newton method to our PDE.

Definition 3.1. *Let X and Y be two Banach spaces, a function $\mathcal{F} : X \to Y$ is said to be Gâteaux differentiable if there exists an operator $T_x : X \to Y$ such that, $\forall \varphi \in X$*

$$\lim_{t \to 0} \frac{\mathcal{F}(x + t\varphi) - \mathcal{F}(x)}{t} = T_x \varphi.$$

The operator T_x is called Gâteaux Derivative of $\mathcal{F}$ at x.

To apply the Newton method to the problem 1.2, we define the operator

$$\mathcal{L}(u) := -\nu \Delta u + h(u)u - f,$$

on which we want to compute the Gâteaux derivative. In fact, similarly to scheme 2.2,

the Newton method reads:

$$D\mathcal{L}_u(\delta u) = -\mathcal{L}(u_k), \tag{3.2}$$

$$u_{k+1} = u_k + \delta u, \tag{3.3}$$

where $D\mathcal{L}$ is indeed the Gâteaux derivative of $\mathcal{L}$, which we compute as follows:

$$D\mathcal{L}_u(\delta u) := \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \left(\mathcal{L}(u_k + \varepsilon\, \delta u) - \mathcal{L}(u_k)\right).$$

This, put in the scheme 3.2, brings us to the linearized sequence of equations, to be solved for δu:

$$-\nu\Delta\delta u + h(u_k)\, \delta u + h'(u_k)u_k\, \delta u = f - (-\nu\Delta u_k + h(u_k)u_k).$$

With the linearized strong form, we can proceed to solve it passing to the weak form

$$\int_\Omega \nabla\delta u \cdot \nabla v + \int_\Omega h(u_k)\, \delta u\, v + \int_\Omega h'(u_k)u_k\, \delta u\, v = -\int_\Omega \mathcal{L}(u_k)v \qquad \forall v \in V$$

and then discretize it applying the Galerking method. In particular, the `fdaPDE` library uses the Finite Element Method as discussed in chapter 1. Introducing suitable finite elements subspaces $V_{\mathtt{h}} \subset V$, the algebraic form of the linearized equation, which represents the linear system we want to solve at each iteration, reads as follows:

$$[A + N_k + N_k']\, \delta u = -\mathcal{L}(u_k), \tag{3.4}$$

$$u_{k+1} = u_k + \delta u, \tag{3.5}$$

where we call A the matrix that comes from the discretization of the Laplacian and N the discretization matrix of the nonlinear term. Note that both matrices N_k and N_k' have to be informed and re-discretized at every iteration.

As we discussed in chapter 2, Newton method has two issues that we should address: the unavailability of h' in analytical form in practical applications and the local convergence property. For what concerns the first issue, we cannot expect the user to provide the derivative of the function h in closed form, and this would be inconvenient for implementation reasons too. Instead, we decided to approximate the function h' using finite differences. Note that, despite what we stated in chapter 2, this approach is now feasible, since in this case we are using finite differences on a function $h : V \to V$, which can be seen as a function mapping $\mathbb{R} \to \mathbb{R}$ on every node of the mesh, while assembling the matrix H_k'. In particular, this is much cheaper than trying to estimate the Jacobian matrix for a nonlinear function $F : \mathbb{R}^n \to \mathbb{R}^n$. Addressing the issue of the local properties instead, we can recover the global strategies analysed for nonlinear functions and apply

them to the step 3.5, which becomes

$$u_{k+1} = u_k + \alpha_k \, \delta u$$

with a Line Search approach.

3.2. Nonlinearity at the Algebraic Level

Another completely different approach consists in postponing the issue of the nonlinearity until we reach the algebraic level of the PDE. Only once the algebraic form is obtained, we can exploit the methods analyzed in Chapter 2 to solve the nonlinear system. The first thing we would like to mention is that this strategy forgets about the PDE that has been treated at the strong level, and therefore the physics behind the nonlinear system of equations, while it blindly tries to solve the nonlinear system arising from the PDE, applying the powerful techniques of unconstrained minimization for general nonlinear equations. In particular, the size of the nonlinear system will depend on the choice of the Galerkin method employed in the discretization of the PDE, which means on the number of degrees of freedom of the mesh we are considering. While in the previous section the Newton method is equation-specific, from now on, the discussion holds even if we modify the structure of the nonlinearity. However, for clarity of exposition, we focus again on the equation 1.2.

Without solving the nonlinearity in 1.2, and recalling the theoretical framework of section 1.2, we can rewrite the discretized formulation of the problem as:

$$\sum_j \int_\Omega \nabla\varphi_j \nabla\varphi_i U_j + \sum_j \int_\Omega h\left(\sum_\ell U_\ell \varphi_\ell\right) \varphi_i \varphi_j U_j = \int_\Omega f\varphi_i \qquad \forall\, i = 1, \dots, N,$$

which brings us once again to the nonlinear system of equations 1.11, that can be rewritten as

$$[A + H(\mathbf{U})]\mathbf{U} - \mathbf{F} = 0$$

and interpreted as a nonlinear function $\mathbb{R}^N \to \mathbb{R}^N$. The application of quasi-Newton methods treated in chapter 2 is then straightforward.

Overall, we recap both the advantages and disadvantages of dealing with the nonlinearity at the strong and algebraic levels. The main advantage of solving the nonlinearity at the strong form, is probably its computational superiority when working on meshes with many nodes, or large computational domains. In fact, linearizing the PDE at the

strong level let us avoid the challenge of solving a nonlinear system of equations, which may have thousands of unknowns. Instead, we deal with the relatively simple task of solving linear systems that arise from the discretization of the linearized equation. In addition, if the nonlinearity is described by a single function h as in our case, and we can approximate h' using finite differences, we don't need advanced multi-dimensional quasi-Newton methods such as the Broyden method, therefore addressing the nonlinearity directly in the strong form appears to be a powerful approach. One limitation of such approach might be, as mentioned at the beginning, that all discussions and computations thus far are valid only for the specific case of equation 1.2. Solving the nonlinearity at the strong form necessitates some knowledge of advanced calculus and the implementation of specialized algorithms for the equation of interest. Therefore, dealing with any other kind of nonlinearity, for which the problem is well posed (i.e. Nonlinear-Diffusion or Nonlinear-Advection Equations), would require the derivation of the strong form of the Newton method and its implementation in the `core` module of the `fdaPDE` library. Dealing with the nonlinearity at the algebraic level instead, gives much more flexibility with respect to the variety of equations that can be solved. In fact, a general method developed in Chapter 2, can be called to solve the algebraic form of basically every well posed nonlinear problem that can be reduced in the form 1.11. However, whenever we want to exploit high order Finite Elements, when we require high accuracy using grids that have many degrees of freedom or simply when the computational domain is very large, attacking directly the nonlinear system 1.11 can be computationally prohibitive, as it can have thousands of unknowns, and the approach presented in 3.1 should be the one to go. Moreover, we should mention that, for large scale nonlinear systems (when the dimension is $n > 500$), the literature suggests to rely on methods which are derivative free, such as nonlinear conjugate gradient methods [13] [26] [25].

4 | Code Structure

The `fdaPDE` library is an `R/C++` library which implements a class of methods for the analysis of spatial and functional data observed on complex domains. These methods are based on regression with PDE regularisation which enable taking into account the physics of the considered problem. The library is organised in a modular structure, allowing to work on singular modules in the developing phase, without the need to recompile the whole codebase. In the layered architecture of the library, we can identify three main modules: *core module*, *model abstraction module* and *model implementation module*. The core layer is where basic data types and operations at the groundings of the library are defined. The model abstraction layer contains an abstract notion of statistical model, based on the groundings provided by the core. Lastly, model implementation layer provides the specialization of the abstract notion of model to well-defined classes of problems.

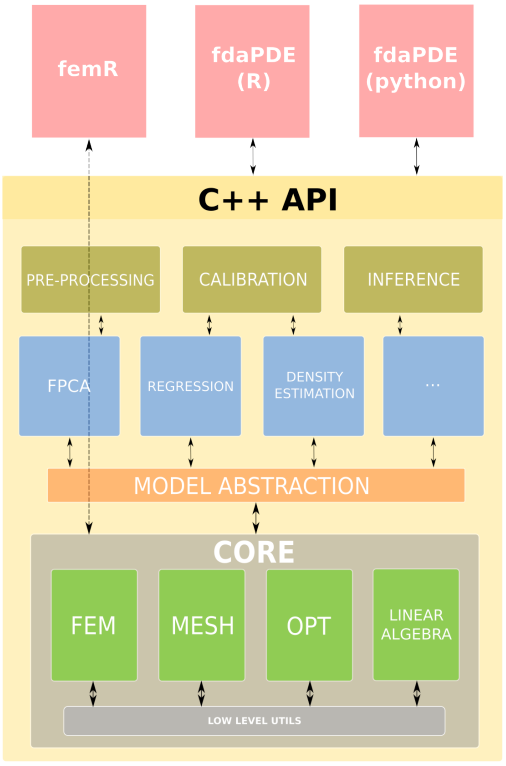

Figure 4.1: Architecture of `fdaPDE` library and `femR` [14].

Solving problems considered in `fdaPDE` requires many steps, at the domain discretization, the discretization of a differential operator, the optimal choice of the smoothing parameter λ and finding the solution of the algebraic system derived from the variational problem. This is why the Core layer is divided into four parts:

- **MESH**: deals with the discretization of the domain.
- **FEM**: takes care of the discretization of the differential operators.
- **OPT**: an ensamble of optimization methods.
- **NLA**: offers numerical linear algebra algorithms.

In this project, we work on the core layer, extending the classes of operators and solvers to handle nonlinear reaction terms. Moreover, we add optimization algorithms to solve more efficiently the algebraic problems. In order to do so, we need to implement the following objects:

- *Field for a nonlinearity.*
- *Nonlinear operator.*
- *Broyden algorithm.*
- *Solvers for nonlinear problems (FixedPoint solver, Newton solver and Broyden solver).*

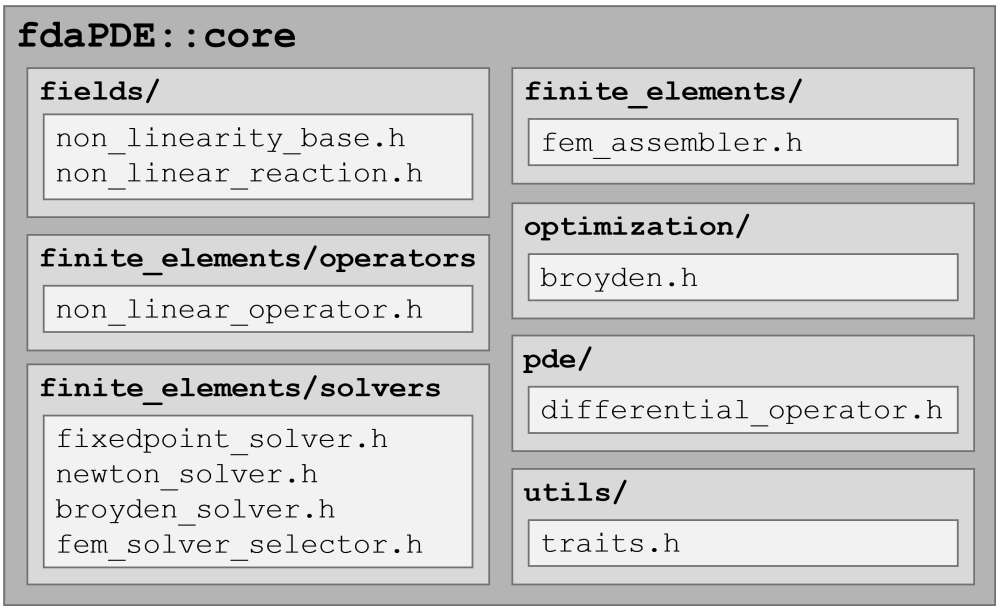

Figure 4.2: Files introduced or modified in the library with this project.

The chapter is organized as follows: in section 4.1 we show the implementation of a

Nonlinear-Reaction term for the assembly of the Finite Elements matrices, in section 4.2 we explain how we wrapped the nonlinearity into a Nonlinear Operator object, in section 4.3 we present the solvers we implemented in the library for solving Nonlinear-Reaction PDEs. Finally, in section 4.4 we discuss how we extended the `R` wrapper of the `femR` library, in order to make our `C++` code accessible from the `R` interface.

4.1. Nonlinear-Reaction

The first step in extending the library to the possibility to solve nonlinear PDEs, is to define a nonlinearity. To do this, we implement a class `NonLinearityBase`, which allows us to define the nonlinear term `h` and to evaluate the solution at the previous step in a point of the domain.

```
template <int N, typename B>
    class NonLinearityBase {
    protected:
        static constexpr std::size_t n_basis_ = B::n_basis;
        typedef std::shared_ptr<DVector<double>> VecP;

        B basis_ {};  //calls default constructor
        mutable VecP f_prev_;  // pointer to vector containing the solution on one
            element at the previous time step
        std::function<double(SVector<N>, SVector<1>)> h = [](SVector<N> x, SVector<1> ff)
             -> double {return 1 - ff[0];};

        // protected method that preforms $\sum_i {f_i*\psi_i(x)}$
        SVector<1> f(const SVector<N>& x) const{
            SVector<1> result;
            result[0] = 0;
            for (std::size_t i = 0; i < n_basis_; i++){
                result[0] += (*f_prev_)[i] * basis_[i](x); }
            return result;
        }

    public:
        // constructor
        NonLinearityBase() = default;
        NonLinearityBase(std::function<double(SVector<N>, SVector<1>)> h_) : h(h_) {}

        //setter for the nonlinear function
        void set_nonlinearity(std::function<double(SVector<N>, SVector<1>)> h_) {h = h_;}
    };
```

`NonLinearityBase` is a template class, and the template parameters represent, respectively, the embedding dimension and the class of the basis for the space where we are looking for the solution. This property allows to define a nonlinearity in any space we want to consider. In order to be able to access the solution at the previous step, we have a member of the class, `f_prev`, that is a pointer to a vector which is declared as `mutable`.

What led us to this implementation choice is the need to re-discretize the differential operator, changing the previous solution at every step, which translates to the need of modifying `f_prev` within the `integrate` method. Because of the way the structure of the library was engineered, the `integrate` methods were already declared as `const`. Therefore, to adapt to the existing structure of the library, we declared `f_prev` as `mutable`. Our objective is to incorporate the nonlinearity into the reaction term. However, rather than limiting the implementation to a singular class for the nonlinear reaction term solely, we opted for a versatile design. This decision is motivated by the possibility of future extensions, allowing for the creation of nonlinearities not only in the reaction term but also in the advection or transport terms. Having defined the base class for a nonlinearity, we build a child class for the nonlinear reaction term which allows to evaluate the nonlinearity in a point in space as well as to modify the pointer to the solution at the previous step.

```
template <int N, typename B>
class NonLinearReaction : public NonLinearityBase<N, B>,
                          public ScalarExpr<N, NonLinearReaction<N, B>>  {
public:
    typedef std::shared_ptr<DVector<double>> VecP;

    auto operator()(VecP f_prev) const{
        this->f_prev_ = f_prev;
        return *this;
    }

    double operator()(const SVector<N>& x) const{
        return this->h(x, this->f(x));
    }
};
```

This class inherits from multiple classes: `NonLinearityBase` and `ScalarExpr`. The latter inheritance uses the *Curiously Recurring Template Pattern (CRTP)* design, an idiom in which a class `X` derives from a class template `Y`, taking as template parameter `X` itself. This implements a compile-time polymorphism, meaning that a base class exposes an interface, and derived classes implement such interface.

If we consider the Newton method applied to the strong form of the PDE, equation 3.2, the derivative of the nonlinearity appears in the discretization too. Hence, we introduce a new class `NonLinearReactionPrime`, which inherits from `NonLinearityBase` and `ScalarExpr`, and has a method for computing the derivative of h through finite differences.

```
template <int N, typename B>
class NonLinearReactionPrime: public NonLinearityBase<N, B>,
                              public ScalarExpr<N, NonLinearReactionPrime<N, B>>  {
public:
    typedef std::shared_ptr<DVector<double>> VecP;
```

```
        double operator()(const SVector<N>& x) const{
            std::function<double(SVector<1>)> lambda_fun = [&] (SVector<1> ff) -> double
                {return this->h(x, ff);};
            ScalarField<1> lambda_field(lambda_fun);
            SVector<1> au;
            au << this->f(x);
            return lambda_field.derive()(au)[0] * this->f(x)[0];
        }

        auto operator()(VecP f_prev) const{
            this->f_prev_ = f_prev;
            return *this;
        }
    };
```

4.2. Nonlinear-Reaction Operator

To solve our problem, we need to introduce a new operator that enables us to discretize the nonlinearity. For this purpose, in line with the library standards, we create the following class:

```
template <typename T>
class NonLinearOp<FEM, T> : public DifferentialExpr<NonLinearOp<FEM, T>>  {
    // perform compile-time sanity checks
    static_assert(
        std::is_invocable<T, std::shared_ptr<DVector<double>>>::value);
   private:
    T h_;
   public:

    // constructor
    NonLinearOp() = default;
    explicit NonLinearOp(const T& h) : h_(h) { }

    // provides the operator's weak form
    template <typename... Args> auto integrate(const std::tuple<Args...>& mem_buffer)
        const {
        IMPORT_FEM_MEM_BUFFER_SYMBOLS(mem_buffer);
        return h_(f) * psi_i * psi_j;
    }
};
```

This class makes use of the *CRTP* as well. In addition, we implemented a `static_assert` declaration to perform compile-time sanity checks. Such strategy has no effect if the argument is well-formed and evaluates to true. Otherwise a compile-time error is issued. This implementation allows us to define the differential operator for our problem in its strong form as follows:

```
    auto L = -nu*laplacian<FEM>() + non_linear_op<FEM>(h);
```

4.3. Solvers

A first important step to solve our problem, is to select the correct solver depending on the PDE. To do this at compile time, first we created a function to understand if we are dealing with a nonlinear PDE.

```
template <typename E_> struct is_nonlinear {
    typedef typename std::decay<E_>::type E;
    static constexpr bool value = has_instance_of_2<NonLinearOp, decltype(std::declval<E
        >().get_operator_type())>::value;
};
```

This is a trait to detect if the differential operator denotes a nonlinear problem that uses `has_instance_of_2`, a recursive template metaprogramming structure that takes two template parameters that we have developed in `utils/traits.h`.

Then, we extended the `fem_solver_selector.h` file, allowing to select a particular solver and setting the pde-dependent Newton method as pre-fixed choice.

```
// selects solver type depending on properties of operator E, carries domain D and
    forcing F to solver
template <typename D, typename E, typename F, typename... Ts> struct pde_solver_selector<
    FEM, D, E, F, Ts...> {
    using type = typename switch_type<
      switch_type_case< is_parabolic<E>::value, FEMLinearParabolicSolver<D, E, F, Ts
          ...>>,
      // switch_type_case<!is_parabolic<E>::value && is_nonlinear<E>::value,
          FEMNonLinearBroydenSolver<D, E, F, Ts...>>,
      // switch_type_case<!is_parabolic<E>::value && is_nonlinear<E>::value,
          FEMNonLinearFixedPointSolver<D, E, F, Ts...>>,
      switch_type_case<!is_parabolic<E>::value && is_nonlinear<E>::value,
          FEMNonLinearNewtonSolver<D, E, F, Ts...>>,
      switch_type_case<!is_parabolic<E>::value && !is_nonlinear<E>::value,
          FEMLinearEllipticSolver<D, E, F, Ts...>>
      >::type;
};
```

Once we are able to formulate our problem, we can implement the solvers. Since we want to solve a nonlinear problem, we have to discretize the differential operator at each step in every solver. To do this, we revisited the class `Assembler`, which deals with the discretization of both the operator and the forcing term.

```
template <typename D, typename B, typename I> class Assembler<FEM, D, B, I> {
   private:
    static constexpr std::size_t n_basis = B::n_basis;
    const D& mesh_;          // triangulated problem domain
    const I& integrator_;    // quadrature rule
    B reference_basis_ {};   // functional basis over reference unit simplex
    int dof_;                // overall number of unknowns in FEM linear system
    const DMatrix<int>& dof_table_;
    DVector<double> f_;   // for non-linear operators, the estimate of the approximated
        solution
```

```
  public:
   Assembler(const D& mesh, const I& integrator, int n_dofs, const DMatrix<int>& dofs) :
     mesh_(mesh), integrator_(integrator), dof_(n_dofs), dof_table_(dofs) {};
   Assembler(const D& mesh, const I& integrator, int n_dofs, const DMatrix<int>& dofs,
       const DVector<double>& f) :
     mesh_(mesh), integrator_(integrator), dof_(n_dofs), dof_table_(dofs), f_(f) {};

   // discretization methods
   template <typename E> SpMatrix<double> discretize_operator(const E& op);
   template <typename F> DVector<double> discretize_forcing(const F& force);

   // setter for f (for non-linear operators)
   void set_f (const DVector<double>& f) {f_ = f;};
};
```

In particular, we introduced a new constructor that takes as parameter the vector of the approximated solution obtained at the previous step, in addition to the other arguments. The discretization of the operator at each step is done using the method `discretize_operator(const E& op)`. We updated this method in a way that, every time it is called, it updates the solution at the previous step in `mem_buffer` in every element of the mesh. We report here a pseudo code of the discretization process of an operator performed by the `Assembler`.

Algorithm 4.1 discretize_operator

1: input: operator `op`
2: prepare the buffer `mem_buffer` to send to the bilinear form
3: develop the bilinear form expression in an integrable field:
   `weak_form = op.integrate(mem_buffer)`
4: **for** `e` : `mesh` **do**
5: **if** `op` is nonlinear **then**
      update `f_` with the solution at the previous step
6: **end if**
7: integrate `op` on the element `e` as in the linear case
8: **end for**

4.3.1. Fixed Point Solver

The Fixed Point solver, from method 3.1, is implemented as a struct inheriting from the class `SolverBase`:

```
   template <typename D, typename E, typename F, typename... Ts>
   struct FEMNonLinearFixedPointSolver : public FEMSolverBase<D, E, F, Ts...>
```

The method `solve` that solves a PDE is defined as follows:

```
template <typename PDE>
void solve(const PDE& pde) {
    static_assert(is_pde<PDE>::value, "pde is not a valid PDE object");

    if(!this->is_init) throw std::runtime_error("solver must be initialized first!");

    // define eigen system solver, use SparseLU decomposition.
    Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>> solver;

    this->solution_.resize(this->n_dofs_,1);

    DVector<double> f_prev;   // solution at the previous step
    f_prev.resize(this->n_dofs_);
    f_prev = pde.initial_condition();  // set initial guess

    // Execute nonlinear loop to solve nonlinear system via fixedpoint method.
    std::size_t i;
    for (i = 0; i < MaxIter_; ++i) {
        // Perform LU decomposition of the system matrix at every step
        solver.compute(this->R1_);                // prepare solver
        if (solver.info() != Eigen::Success) {    // stop if something was wrong...
            this->success = false;
            return;
        }
        this->solution_ = solver.solve(this->force_);   // solve linear system

        // Check convergence to stop early
        auto incr = this->solution_ - f_prev;
        if (incr.norm() < tol_) break;

        f_prev = this->solution_;

        // Update the system matrix for the next iteration.
        Assembler<FEM, D, FunctionSpace, QuadratureRule> assembler(pde.domain(),
        this->integrator_, this->n_dofs_, this->dofs_, f_prev);
        this->R1_ = assembler.discretize_operator(pde.differential_operator());
        this->R1_.makeCompressed();

        // set dirichlet boundary conditions on the system matrix
        this->set_dirichlet_bc(pde);
    }
    this->success = true;
    return;
}
```

At the beginning of the method, we use a `static_assert` declaration to asses at compile-time if the input argument of `solve` is a valid PDE object. The algorithm used here consists in solving at each iteration, with the `Eigen LU` solver, the system $\texttt{R1}u = f$, where `R1` is the matrix in which we have discretized the differential operator and f is the forcing term of the PDE. Then, we check the stopping criterion and, if it is not satisfied, we update the solution at the previous step and discretize again the differential operator. In this algorithm, we chose as stopping criterion $\|\texttt{incr}\| < \texttt{tol}$, where `incr` is the increment

between the current solution and the solution at the previous step, while `tol` is a given tolerance set by the user.

4.3.2. Newton Solver

The Newton solver, from method 3.5, is implemented as a struct inheriting from the class `SolverBase`:

```
    template <typename D, typename E, typename F, typename... Ts>
    struct FEMNonLinearNewtonSolver : public FEMSolverBase<D, E, F, Ts...>
```

The method that solves a PDE is defined below.

```
template <typename PDE>
    void solve(const PDE& pde) {
        static_assert(is_pde<PDE>::value, "pde is not a valid PDE object");

        if(!this->is_init) throw std::runtime_error("solver must be initialized first!");

        // first solve the linear associated pde (initial guess = 0)
        DVector<double> f_prev = DVector<double>::Zero(this->n_dofs_);
        Assembler<FEM, DomainType, ReferenceBasis, Quadrature> assembler(pde.domain(),
            this->integrator_, this->n_dofs_, this->dofs_, f_prev);
        SpMatrix<double> A = assembler.discretize_operator(pde.differential_operator());
        A.makeCompressed();
        // set B.C.s
        for (auto it = this->boundary_dofs_begin(); it != this->boundary_dofs_end(); ++it
            ) {
            A.row(*it) *= 0;            // zero all entries of this row
            A.coeffRef(*it, *it) = 1;   // set diagonal element to 1 to impose equation u
                _j = b_j
        }
        // define eigen system solver, use SparseLU decomposition.
        Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>> solver;
        solver.compute(A);
        if (solver.info() != Eigen::Success) {    // stop if something went wrong...
            this->success = false;
            std::cout << "LU return due to success=false" << std::endl;
            return;
        }
        f_prev = solver.solve(this->force_);

        this->solution_.resize(this->n_dofs_,1);

        // declare the known term that will go on the right hand side and will contain h'
        NonLinearReactionPrime<DomainType::local_dimension, ReferenceBasis> h_prime;
        auto Lprime = non_linear_op<FEM>(h_prime);

        // Execute nonlinear loop to solve nonlinear system
        std::size_t i;
        for (i = 0; i < MaxIter_; ++i) {
            // re-declare the assembler with f_prev updated.
            Assembler<FEM, DomainType, ReferenceBasis, Quadrature> assembler(pde.domain()
                , this->integrator_, this->n_dofs_, this->dofs_, f_prev);
```

```
        // discretize stiffness matrix
        this->R1_ = assembler.discretize_operator(pde.differential_operator());
        this->R1_.makeCompressed();
        // set dirichlet boundary conditions on the system matrix
        this->set_dirichlet_bc(pde);

        // discretize derived nonlinearity
        SpMatrix<double> R2 = assembler.discretize_operator(Lprime);
        R2.makeCompressed();
        for (auto it = this->boundary_dofs_begin(); it != this->boundary_dofs_end();
            ++it) {
            R2.row(*it) *= 0;            // zero all entries of this row
            R2.coeffRef(*it, *it) = 1;   // set diagonal element to 1 to impose
                equation u_j = b_j
        }

        // Perform LU decomposition of the system matrix at every step
        solver.compute(this->R1_ + R2);                // prepare solver
        if (solver.info() != Eigen::Success) {    // stop if something was wrong...
            this->success = false;
            std::cout << "Return due to success=false at iteration " << i << std::
                endl;
            return;
        }

        // Newton direction
        DVector<double> sN = solver.solve(this->force_ - this->R1_*f_prev);
        this->solution_ += sN;

        // Check convergence to stop early
        auto incr = this->solution_ - f_prev;
        std::cout << "Newton solver: norm of the increment at iteration " << i << " 
            is " << incr.norm() << std::endl;
        if (incr.norm() < tol_) break;

        f_prev = this->solution_;
    }
    this->success = true;
    return;
}
```

The algorithm we implemented in this method uses as initial guess the solution of the linear system associated with our problem, and solves at each iteration, using again the `Eigen LU` solver, the linear system $(\texttt{R1} + \texttt{R2})u = f$. `R2` is the matrix coming from the discretization of the derivative of the nonlinear term while `R1` and f are defined as before. At each iteration we need to discretize again both matrices `R1`, `R2`. For this algorithm as well, we choose as stopping criterion $\|\texttt{incr}\| < \texttt{tol}$.

4.3.3. Broyden

In order to build a Broyden solver, we first need to implement the Broyden method as an optimization algorithm that solves a nonlinear system $F(x) = 0$, where $F : \mathbb{R}^n \to \mathbb{R}^n$. To do this, we create a `struct` with a different method for each variation of the Broyden algorithm we considered and implement.

```
template <int N>
struct Broyden {
   private:
    typedef typename std::conditional<N == Dynamic, DVector<double>, SVector<N>>::type
        VectorType;
    typedef typename std::conditional<N == Dynamic, DMatrix<double>, SMatrix<N>>::type
        MatrixType;
    std::size_t max_iter_ = 500;   // maximum number of iterations before forced stop
    double tol_ = 1e-3;            // tolerance on error before forced stop
    std::size_t nmax_ = max_iter_ + 1;

   public:
    // constructor
    Broyden() = default;
    Broyden(std::size_t max_iter, double tol) : max_iter_(max_iter), tol_(tol) {};
    Broyden(std::size_t max_iter, std::size_t nmax, double tol) : max_iter_(max_iter),
        nmax_(nmax), tol_(tol) {};

    // solvers
    template <typename F> VectorType solve(F&, VectorType&) const;
    template <typename F> VectorType solveArmijo(F&, VectorType&) const;
    template <typename F> VectorType solve_modified(F&, VectorType&) const;
    template <typename F> VectorType solve_modified_inv(F&, VectorType&) const;
    template <typename F> VectorType solve_trust_region(F&, VectorType&);
};
```

The type definition through `std::conditional` makes the code more flexible since, depending on the template parameter, `VectorType` represents both a fixed dimensional and a dynamic vector.
In this class, we implemented all the Broyden algorithm presented in Chapter 2. In particular, the `solve` method implements the good Broyden method, then we extended it in `solveArmijo` adding a Line Search strategy with the Armijo condition. The `solve_modified` method, instead, implements the modified Broyden-like method with Line Search as in [26], while in `solve_modified_inv`, we implemented a method with the same Line Search technique, but using the bad Broyden update. Lastly, we developed the method `solve_trust_region` that uses the trust region technique and the good Broyden update.

The algorithm that we selected for solving our problem is the global version of the modified Broyden-Like method, from algorithm 2.1, and it is implemented in the `solve_modified` method of the Broyden struct.

```
    template <typename F>
```

```
VectorType solve_modified(F& f_, VectorType& x) const {
    static_assert(
        std::is_same<decltype(std::declval<F>().operator()(VectorType())),
        VectorType>::value,
        "cannot␣find␣definition␣for␣F.operator()(const␣VectorType&)");

    int dim = x.size(); // = f_(x).size();
    VectorType d_B(dim), d_MB(dim), d(dim);
    VectorType s(dim), y(dim);
    MatrixType B = DMatrix<double>::Identity(dim, dim);

    int itc = 0;   // current iteration

    // parameters
    double tau = 1.;
    double rho = sqrt(0.9);
    double alfa_k = 1.;
    double theta_k = 1.;
    double r = 0.1;
    auto eps = [](int n) -> double {return 1./((n+1)*(n+1));};
    double sigma_1 = 0.1;
    double sigma_2 = 0.01;
    bool step_5 = false;

    double norm_fx = 0;
    VectorType fx(dim);
    double norm_fz = 0;
    VectorType fz(dim);

    while (itc < max_iter_){
        itc += 1;
        step_5 = false;
        fx = f_(x);
        norm_fx = fx.norm();

        // step 2
        if (norm_fx < tol_) {return x;}

        // step 3
        // compute s_B_n+1
        Eigen::GMRES<DMatrix<double>> solver(B);
        d_B = solver.solve(-fx);   // solve linear system

        auto z = x + d_B;
        fz = f_(z);
        norm_fz = fz.norm();

        if (norm_fz > tau*(norm_fx)){
            d = d_B;
            // go to step 4
        }
        else {
            d_MB = solver.solve(-fz);
            if (f_(x + d_B + d_MB).norm() <= rho*norm_fx - sigma_1*(d_B + d_MB).
                squaredNorm()){
                d = d_B + d_MB;
```

```
                alfa_k = 1.;
                // go to step 5
                step_5 = true;
            }
            else d = d_B; // go to step 4
        }

        // step 4
        if (step_5 == false){
            if (f_(x+d).norm() <= rho*norm_fx - sigma_1*d.squaredNorm())
                alfa_k = 1.;
            else {
                alfa_k = 1.;
                while (f_(x+alfa_k*d).norm() > norm_fx - sigma_2*(alfa_k*d).
                    squaredNorm() + eps(itc)*norm_fx)
                    alfa_k *= r;
            }
        }

        // step 5
        auto x_new = x + alfa_k*d;

        // step 6
        // compute B_n+1
        s = x_new - x;
        y = f_(x_new) - fx;
        B = B + theta_k * (y - B*s)*s.transpose() / s.squaredNorm();

        x = x_new;
    }
    return x;
}
```

The Broyden solver is implemented as a `struct` inheriting from the base class, and uses the above algorithm to solve the problem 2.1 where $F = Au - f$, being A the matrix in which we discretized our differential operator and u the forcing term.

```
template <typename D, typename E, typename F, typename... Ts>
struct FEMNonLinearBroydenSolver : public FEMSolverBase<D, E, F, Ts...>
```

The `solve` method that solves a PDE is defined below.

```
template <typename PDE>
void solve(const PDE& pde) {
    static_assert(is_pde<PDE>::value, "pde␣is␣not␣a␣valid␣PDE␣object");

    if(!this->is_init) throw std::runtime_error("solver␣must␣be␣initialized␣first!");

    // first solve the linear associated pde (initial guess = 0)
    DVector<double> f_prev = DVector<double>::Zero(this->n_dofs_);
    // Assembler<FEM, D, FunctionSpace, QuadratureRule> assembler(pde.domain(), this
        ->integrator_, this->n_dofs_, this->dofs_, f_prev); //old
    Assembler<FEM, DomainType, ReferenceBasis, Quadrature> assembler(pde.domain(),
        this->integrator_, this->n_dofs_, this->dofs_, f_prev);
    SpMatrix<double> A = assembler.discretize_operator(pde.differential_operator());
```

```
        A.makeCompressed();
        // set B.C.s
        for (auto it = this->boundary_dofs_begin(); it != this->boundary_dofs_end(); ++it
            ) {
            A.row(*it) *= 0;            // zero all entries of this row
            A.coeffRef(*it, *it) = 1;   // set diagonal element to 1 to impose equation u
                _j = b_j
        }

        Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>> solver;
        solver.compute(A);
        if (solver.info() != Eigen::Success) {    // stop if something went wrong...
            this->success = false;
            std::cout << "LU return due to success=false" << std::endl;
            return;
        }
        DVector<double> x0 = solver.solve(this->force_);

        // F(u) = A(u)*u - b
        std::function<DVector<double>(DVector<double>)> Fun;

        Fun = [&](DVector<double> u) -> DVector<double> {
            // A(u) * u - f = 0
            this->solution_ = u;
            this->init(pde);
            this->set_dirichlet_bc(pde);
            return this->R1_ * u - this->force_;
        };

        // solve with Broyden Line Search
        Broyden<Dynamic> br(MaxIter_, tol_);
        this->solution_ = br.solve_modified(Fun, x0);

        this->success = true;
        return;
    }
```

In order to apply the Broyden algorithm to our problem, we first compute the solution of the linear associated PDE, as we did for the Newton method. Then, using such solution as an initial guess, we apply the Broyden algorithm with Line Search to system 2.1. This solver uses as stopping criterion the one from the Broyden algorithm, that, for the considered function F, is the norm of the residual.

4.4. R wrapper

This section introduces the main changes to the source code of the `R` package `femR` to incorporate the new functionalities developed in this project.

The first step in building the `R` interface for the nonlinear case is extending the existing `R` wrapper to handle Elliptic Nonlinear-Reaction Partial Differential Equations. We thus

modify the file `r_pde.h` contained in the `femR` module of the library [8], which we report here:

```
template <int M, int N, int R> class R_PDE : public PDEWrapper {
   private:
    using DomainType = Mesh<M, N>;
    using QuadratureRule = Integrator<FEM, DomainType::local_dimension, R>;
    using ReferenceBasis = typename LagrangianBasis<DomainType, R>:: ReferenceBasis;
    template <typename L> using PDEType = PDE<DomainType, L, DMatrix<double>, FEM, fem_
        order<R>>;

    DomainType domain_ {};           // triangulation
    QuadratureRule integrator_ {};   // quadrature rule
    int pde_type_;                   // one of the pde_type enum values
    std::function<double(SVector<N>, SVector<1>)> h_;  // added
   public:
    // constructor
    R_PDE(Rcpp::Environment mesh, int pde_type, const Rcpp::Nullable<Rcpp::List>& pde_
        parameters) :
        pde_type_(pde_type){
        // set domain
        ...
        // unpack pde parameters
        ...
        // instantiate pde template based on pde type
        switch (pde_type_) {
        case pde_type::simple_laplacian: {...} break;
        case pde_type::second_order_elliptic: {...} break;
        case pde_type::second_order_parabolic: {...} break;
        case pde_type::non_linear_reaction: {
            SMatrix<M> K = Rcpp::as<DMatrix<double>>(pde_parameters_["diffusion"]);
            DVector<double> coeffs = Rcpp::as<DVector<double>>(pde_parameters_["binomial"
                ]);
            this -> h_ = [coeffs](SVector<N> x, SVector<1> ff) -> double {return coeffs
                [0]*(coeffs[1] + coeffs[2]*ff[0]);};
            NonLinearReaction<N, ReferenceBasis> non_linear_reaction_(this->h_);
            auto L = diffusion<FEM>(K) + advection<FEM>(b) + reaction<FEM>(c) + non_
                linear_op<FEM>(non_linear_reaction_);
            pde_ = PDEType<decltype(L)>(domain_, L, this->h_);
        } break;
     }
    }
    ...
};
```

In particular, the constructor now handles the case of a full Elliptic Nonlinear-Reaction PDE, therefore, from the `R` interface, it will be possible to specify a nonlinear reaction term of the form

$$\alpha(\beta + \gamma u)u$$

where the rparameters α, β, γ can be chosen by the user. Once again, as stated at the beginning of our work, with the `femR` library we want to give to the end-user the freedom to instantiate a full PDE containing diffusion, transport, linear and nonlinear reaction

terms. It will be the user's responsibility to check the well posedness of the mathematical problem they want to solve before implementing it in `femR`.

At the `R` level instead, we create two new classes that extend the `R` interface by inheriting from the `.DifferentialOpCtr` class. One of them is the `.NonLinearReaction` class, which we use to instantiate a Nonlinear-Reaction object. The other is the `binomial` class, which we need to write the block $(1-u)$ in the `R` interface.

```
.NonLinearReaction <- R6Class("NonLinearReaction",
                              inherit = .DifferentialOpCtr)

.Binomial <- R6Class("Binomial", inherit = .DifferentialOpCtr,
                     public = list(
                       initialize = function(params = c(1.,1.,1.), Function){
                         super$initialize(tokens = "binomial",
                                          params = list(params),
                                          Function = Function)
                       }))
```

Furthermore, along these classes, we implement suitable operators as well. By doing so, we can write in `R` any block of the form `(k+u)`, `(k-u)` and their symmetrical counterparts `(u+k)`, `(u-k)`, where `k` is a generic constant of type `numeric` and `u` is a `Function` class.

```
`-.Function` <- function(e1, e2){
  if(is(e1, "numeric") & is(e2, "Function"))
    .Binomial$new(params = c(1.,1.,-1), Function = e2)
  else if(is(e2, "numeric") & is(e1, "Function"))
    .Binomial$new(params = c(1.,-1.,1.), Function = e1)
}

`+.Function` <- function(e1, e2){
  if(is(e1, "numeric") & is(e2, "Function"))
    .Binomial$new(Function = e2) #
  else if(is(e2, "numeric") & is(e1, "Function"))
    .Binomial$new(Function = e1)
```

For the `R` interface to work correctly, we also modify the `*` operator for the `.DifferentialOp` class, as we need to write the nonlinear-reaction term `alpha*u(1-u)` by multiplying a `binomial` object, represented by `(1-u)`, with a `ReactionOperator` object, represented by `alpha*u`. The next script shows the changes we made.

```
`*.DifferentialOp` <- function(e1,e2){
            if (is.numeric(e1) & !is(e2, "Binomial")){
              e2$params[[1]] <- e1*e2$params[[1]]
              e2
            }else if(is.numeric(e1) & is(e2, "Binomial")){
              e2$params[[1]][1] <- e1*e2$params[[1]][1]
              e2
            }else if( (is(e1, "Binomial") & is(e2, "ReactionOperator"))){
              .NonLinearReaction$new(
                        params=list(
                            binomial = c(e1$params[[1]][1]*e2$params[[1]],
```

```
                                        e1$params[[1]][2:length(e1$params[[1]])]),
                            tokens = "non_linear_reaction",
                            Function=e1$Function()
      )
    }else if((is(e2, "Binomial") & is(e1, "ReactionOperator"))){
      .NonLinearReaction$new(
                  params=list(
                            binomial = c(e2$params[[1]][1]*e1$params[[1]],
                                        e2$params[[1]][2:length(e2$params[[1]])])),
                            tokens = "non_linear_reaction",
                            Function=e1$Function())
    }else
      stop("Wrong input arguments.")
}
```

Whit these modifications to the R wrapper, we are able to write the differential operator of the Nonlinear-Reaction PDE 1.2 in its strong form as follows:

```
DifferentialOp <- -mu*laplace(u) + alpha*u*(1-u)
```

In this way, the R interface remains clear, and implementing a Nonlinear-Reaction PDE is straightforward and user friendly.

5 | Numerical Results

In this chapter, we focus on the results obtained while solving problem 1.2 with the code we implemented. In section 5.0.1 we compare the results of our `C++` code both with the expected theoretical results, and with existing softwares for solving Partial Differential Equations using `C++`, in particular we consider the `FreeFem++` [9] library as a comparison benchmark. In section 5.0.2 instead, we embed the code we implemented at the `fdaPDE/core` layer into the complete modular structure of `femR`. Running the code from the `R` interface, we compare our results with the only tool available (up to our knowledge) in `R` for solving Partial Differential Equations, which is `deSolve` [22].

5.0.1. Numerical Results at the `core` layer in `C++`

For the test phase, we consider the problem 1.2 where we set $\nu = 1$ and $\alpha = 1$, not to incur in numerical instabilities due to reaction-dominated phenomena, while the domain Ω is the 2D unit square $[0, 1]^2$. Moreover, for testing purposes, we put ourselves in a friendly framework in which we define the exact solution, and we compute the forcing term accordingly. In particular, we choose the function $u_{\text{ex}} \in H_0^1$ and of class $\mathcal{C}^2$ to be

$$u_{\text{ex}}(x, y) = 3x^2 + 2y^2,$$

and the forcing term for the equation at hand is computed accordingly

$$f(x, y) = -9x^4 - 12x^2y^2 + 3x^2 - 4y^4 + 2y^2 - 10.$$

Therefore, the particular problem we are considering in the test phase reads as follows:

$$\begin{cases} -\Delta u + (1 - u)u = -9x^4 - 12x^2y^2 + 3x^2 - 4y^4 + 2y^2 - 10 & \text{in } [0, 1]^2 \\ u = u_{\text{ex}} & \text{on } \partial\left([0, 1]^2\right) \end{cases} \tag{5.1}$$

For the following tests, the computational domain for the Finite Element Method consists in a discretization of the unit square with 32 triangles per side, represented in figure 5.1.

The Finite Element order has been chosen to be $r = 2$.

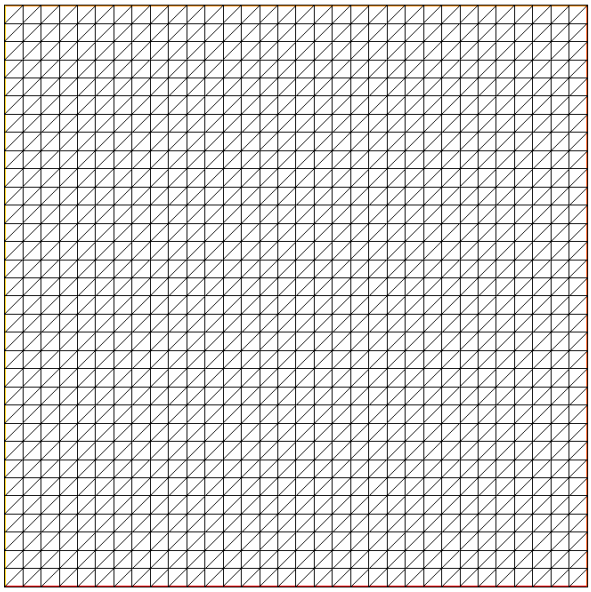

Figure 5.1: Computational mesh `unit_square_32` used for testing.

We run the first test to assess the correctness of our Fixed Point method. Considering the problem 5.1, we solve it using the Fixed Point Solver we implemented, taking as initial guess the null function $u_0 = 0$. At each iteration, we compute the L^2-norm of the error between the current solution and the reference solution and we save such value in a vector. Setting the tolerance to be $\texttt{tol} = 1 \times 10^{-7}$, the Fixed Point Solver takes 4 iterations to lower the error below the tolerance and, in particular, the final error is of the order 1×10^{-9}.

Using the vector containing the history of the error, we can estimate the order of convergence q of the Fixed Point method, with the formula

$$q \approx \frac{\log(e_{k+1}/e_k)}{\log(e_k/e_{k-1})}, \tag{5.2}$$

where $e_k = \|u_k - u_{\text{ex}}\|_2$ [20]. In particular, the estimated order for the Fixed Point method is $q = 1.002$, indicating a linear convergence rate, as expected. Finally, in figure 5.2 we plot the error at every iteration, together with a linear decrease reference.

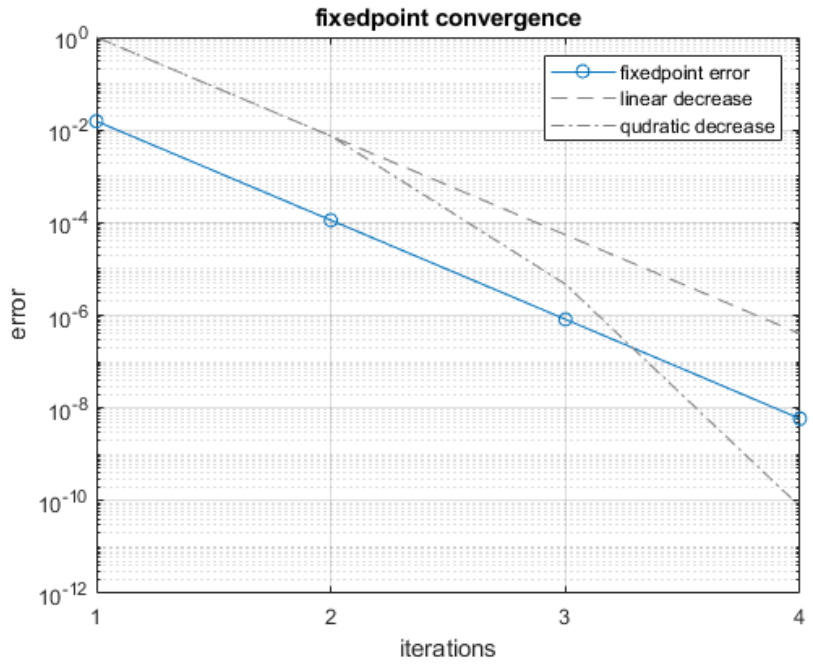

Figure 5.2: Analysis of the error of the Fixed Point method, which yields a linear convergence rate.

The graphical representation gives yet another validation of the linear convergence rate of the Fixed Point Solver implemented.

After testing the Fixed Point method, we proceed to assess the performance of the Newton method. The solution to problem 5.1 is now obtained through the application of the Newton method to the strong formulation of the PDE, as outlined in chapter 3 and further developed in chapter 4. For what concerns the initial guess for the Newton Solver, we first perform one iteration of the Fixed Point Solver, again with an initial guess $u_0 = 0$, as this is equivalent to solve the associated linear equation. The outcome of the first iteration is then used as a starting point for the Newton Solver. Again, setting the tolerance to `tol` $= 1 \times 10^{-7}$, the Newton Solver takes 3 iterations (linear equation + 2 Newton iterations) to lower the error below the tolerance and, in particular, the final error is of the order 1×10^{-15}. Using formula 5.2, we can estimate the order of convergence of the Newton method as well, which turns out to be $q = 1.972$, indicating a quadratic convergence rate, as expected. Finally, in figure 5.3, we plot the error at every iteration for the Newton method. The graphical representation confirms once again the analytical estimate.

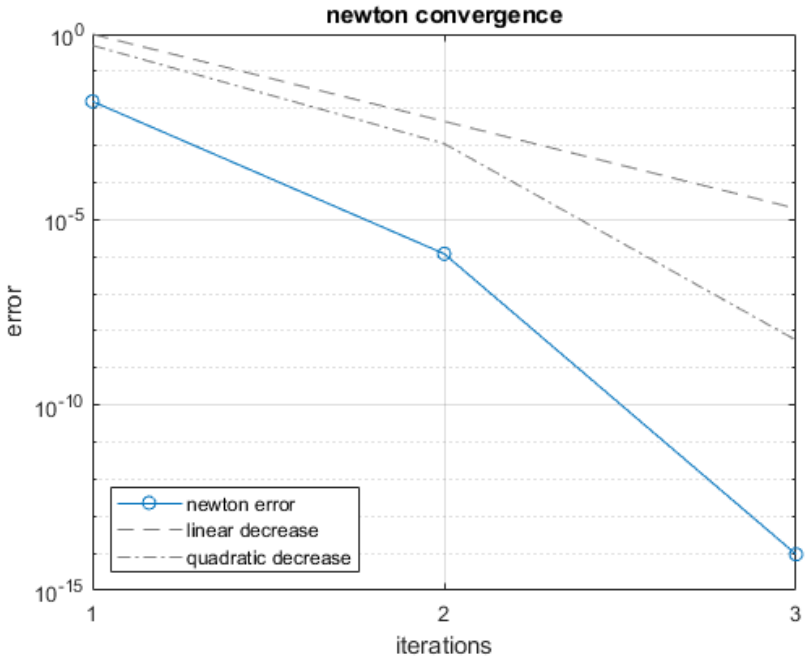

Figure 5.3: Analysis of the error of the Newton method, which yields a quadratic convergence rate.

After testing both the Fixed Point and the Newton solvers, we investigate the performance of the Broyden method for solving the nonlinear system of equation 1.11. As we mentioned in chapter 3, the literature suggests using such method when the size of the nonlinear system is not too large. In particular, the work done in [26] suggest the size n of the nonlinear system to be at most $n \approx 500$. For this reason, we first apply the Broyden method to the nonlinear system of equations 1.11 with a smaller computational domain, that has 16 triangles per side, shown in figure 5.4, and a Finite Element linear order $r = 1$.

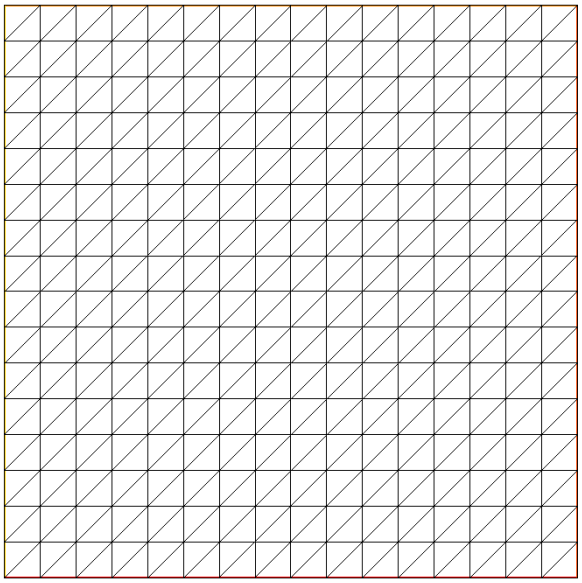

Figure 5.4: Computational mesh `unit_square_16` used for testing.

With this choice, the number of degrees of freedom (`ndofs`) in the mesh, and therefore the size of the nonlinear system, is `ndofs` = 289. In figure 5.5 we plot the error, computed as for the previous methods, versus the number of iterations:

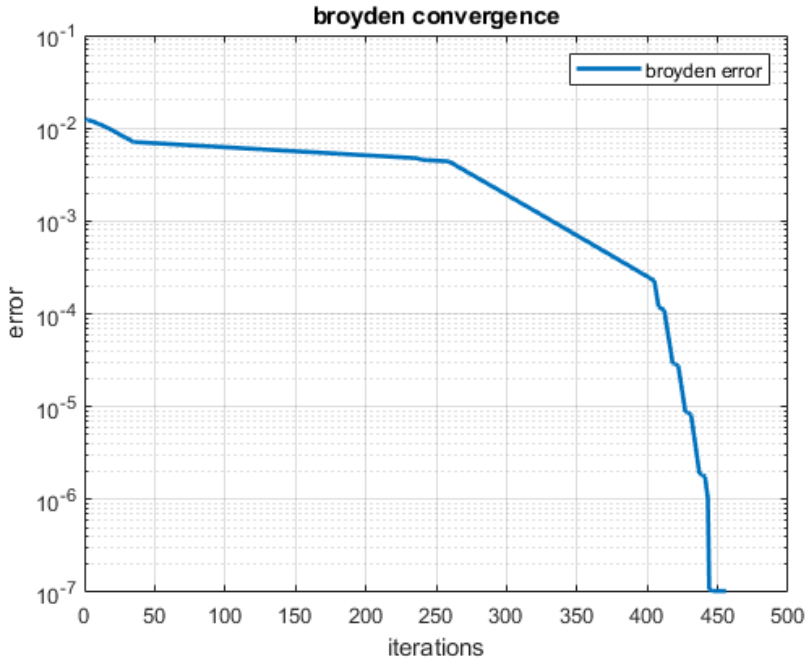

Figure 5.5: Analysis of the error of the Broyden method.

A first observation is that the Broyden method takes a significantly larger amount of

iterations (456) to bring the error below the tolerance $\texttt{tol} = 1 \times 10^{-7}$. Our explanation for this result is that the Broyden method does not take into account the PDE and only solves the nonlinear system that arises from the equation's algebraic form. This is a very general way to deal with any nonlinearity, but the PDE-dependent scheme is preferable when available, because it is more effective. It is also evident that, despite the method's global nature, it performs very poorly and has a weak convergence rate throughout most of the iterations, when the Line Search strategy is most needed, as it makes little advancement towards the solution. As we approach the exact solution instead, global strategies are not needed anymore, and the method gains back a superlinear convergence rate.

Having analyzed the Broyden method in the most simple scenario, when we try to increase the number of Finite Elements in our grid, or to switch back from linear to quadratic Finite Elements, the number of degrees of freedom, and therefore the size of the nonlinear system, becomes too large for the Broyden method to give reasonable results. We show in figure 5.6 the decrease of the error when we solve problem 5.1 with Broyden method using a computational domain that has 16 triangles per side and the Finite Element order has been increased to $r = 2$. In particular, in this case, $\texttt{ndofs} = 1089$ and the solver takes 1598 iterations to lower the error below the given tolerance $\texttt{tol} = 1 \times 10^{-7}$.

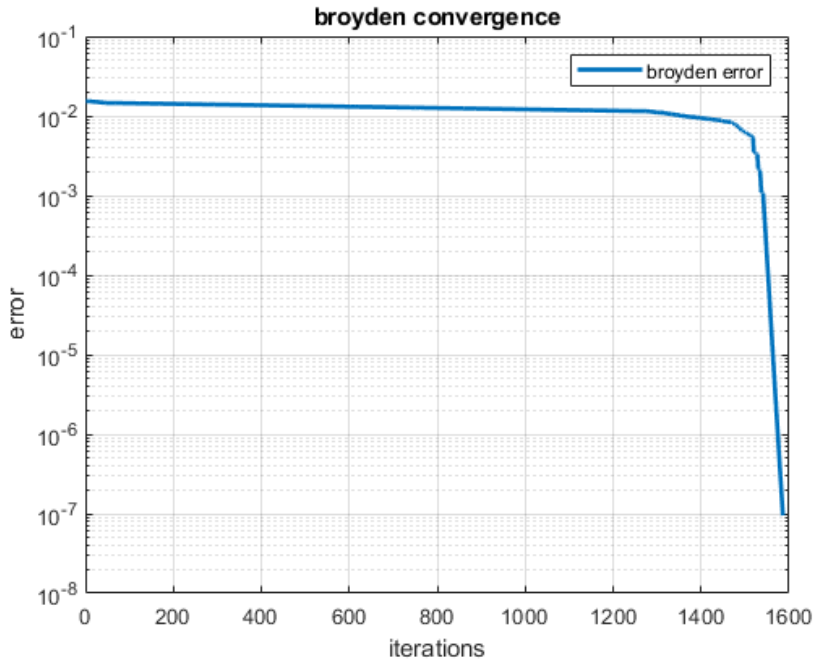

Figure 5.6: Analysis of the error of the Broyden method. Size of the nonlinear system $n = 1089$.

It is evident that the Broyden method, although theoretically applicable to any nonlin-

earity, performs poorly and becomes computationally expensive when dealing with large scale problems.

As a conclusive outcome for this section, we compare the performance of our Fixed Point and Newton solvers with the corresponding techniques implemented in `FreeFem++` for solving problem 5.1. We refer the reader to the appendix A for the full `FreeFem++` code implementation. The following histogram shows the average computational time over multiple instances of the numerical tests of both solvers using the code developed in `femR` and `FreeFem++`.

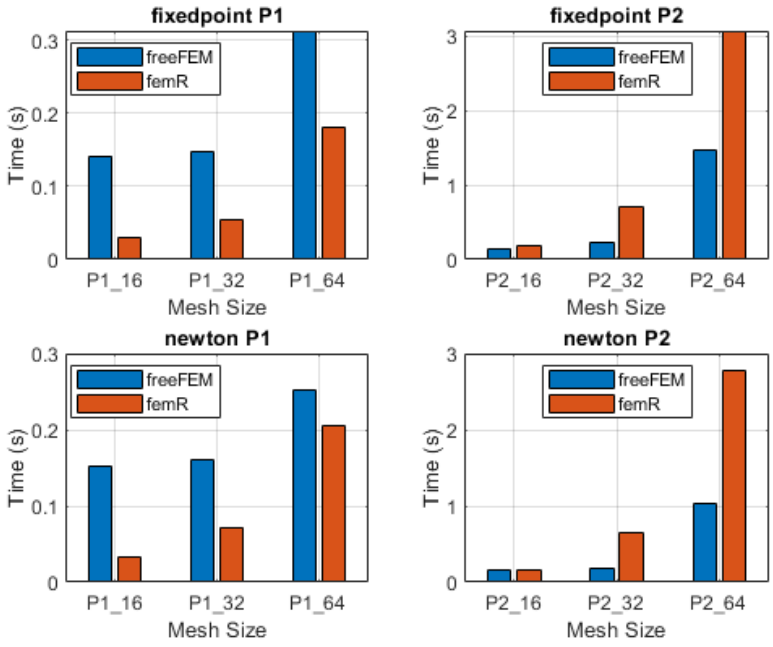

Figure 5.7: Performanceof our `C++` code versus `FreeFem++` code.

As it can be noticed, our computational times are on the same order of magnitude with respect to the `FreeFem++` library.

5.0.2. Numerical Results of `femR`

After assessing the correctness and testing the performance of the `C++` code we implemented, using `FreeFem++` as a comparison benchmark, we proceed in implementing the wrapper for the `R` interface. In this section, we shall compare the results we obtain using the `femR` library within the `R` environment. As mentioned earlier, the comparison benchmark for us in this case will be the `deSolve` package [22].

For the comparison of the codes in `R`, following the example presented in [21], we build

the R script to solve problem 5.1, that we considered in the previous subsection, with the deSolve package. We report here an extract of such script:

```
# forcing term
f <- function(points){...}
# exact solution
exact <- function(points){...}

# mesh generation (finite differences)
x.grid    <- setup.grid.1D(x.up = 0, x.down = 1, N = N)
y.grid    <- setup.grid.1D(x.up = 0, x.down = 1, N = N)
grid2D    <- setup.grid.2D(x.grid, y.grid)
h = max(grid2D$dx, grid2D$dy)
D.grid    <- setup.prop.2D(value = mu, y.value = mu, grid = grid2D)
v.grid    <- setup.prop.2D(value = 0, grid = grid2D)
A.grid    <- setup.prop.2D(value = 1, grid = grid2D)
VF.grid   <- setup.prop.2D(value = 1, grid = grid2D)
C.x.up    <- exact(cbind(rep(x.grid$x.up, times=N), y.grid$x.mid))
C.y.up    <- exact(cbind(x.grid$x.mid, rep(y.grid$x.up, times=N)))
C.x.down  <- exact(cbind(rep(x.grid$x.down, times=N), y.grid$x.mid))
C.y.down  <- exact(cbind(x.grid$x.mid, rep(y.grid$x.down, times=N)))

# populate forcing & uex
forcing = matrix(nrow=N, ncol=N, data=0)
u.ex = matrix(nrow=N, ncol=N, data=0)
for(k in 1:N){
  for(l in 1:N){
    forcing[k,l] = f(cbind(grid2D$x.mid[k], grid2D$y.mid[l]))
    u.ex[k,l] = exact(cbind(grid2D$x.mid[k], grid2D$y.mid[l]))
  }
}
# define the problem
Diff2Db <- function (t, y, parms)  {
  Y  <- matrix(nrow = N, ncol = N, data = y)
  dY <- tran.2D(C = Y,
                C.x.up = C.x.up, C.x.down = C.x.down,
                C.y.up = C.y.up, C.y.down = C.y.down,
                grid = grid2D,
                D.grid = D.grid,
                A.grid = A.grid,
                VF.grid = VF.grid,
                v.grid = v.grid)$dC
  dY <- dY - alpha*Y*(1-Y) + forcing
  return (list(dY))
}
y = matrix(data = rep(1,times=N*N))
y = matrix(data = rnorm(N*N, mean = 1, sd = 0.5))

# solve the problem
Y <- steady.2D( y=y, dimens = c(N,N),
    time = 0, func = Diff2Db,
    parms=NULL, lrw=1e8 )
u.deSolve <- matrix(Y$y, nrow=N, ncol=N)
```

Probably, the first thing that stands out, is that such script is quite involved and very little

user friendly. A non-expert user that approaches a similar problem in R would require some time to get acquainted with the syntax and structure of the deSolve package.

Once the benchmark script with deSolve is built, we proceed writing the code to solve the very same problem using the femR library, taking advantage of the R wrapper for the code implemented in the fdaPDE/core submodule. We report here an extract of the femR implementation for solving problem 5.1:

```
library(femR)
# forcing term
f <- function(points){...}
# exact solution
exact <- function(points){...}

mesh <- fdaPDE::create.mesh.2D(grid2D)
nnodes <- nrow(mesh$nodes)
square <- list(nodes= mesh$nodes, elements= mesh$triangles, boundary= mesh$nodesmarkers)
mesh <- Mesh(square)

Vh <- FunctionSpace(mesh, 1L)
u <- Function(Vh)
DifferentialOp <- -mu*laplace(u) + alpha*u*(1-u)

pde <- Pde(DifferentialOp = DifferentialOp,
         forcing = f,
         boundary_condition = exact)

# solve problem
pde$solve()
```

The greatest advantage in using the femR library is certainly the simplicity of the interface, which is more intuitive, plain and understandable, even for a non-advanced R user. In particular, despite what is required with the deSolve package, while defining the differential problem, the user doesn't have to worry about the iterative method that will be used by femR during the solution process. It is sufficient to declare a differential operator and build a PDE object, specifying a forcing term and boundary conditions.

At last, in this section, we show the results of the comparison between the two R scripts we implemented, and comment on them. To do so, we run both the scripts in a for loop, with an increasing size of the mesh given by the vector N=c(16, 32, 64). It's worth mentioning that, while femR uses a Finite Elements approach, this is not the case for the deSolve package. The latter implements instead an iterative scheme based on finite differences, using a parameter h. In particular, this discrepancy brings yet another advantage in the use of the femR library when the differential problem is defined on a complex domain, for which unstructured meshes are necessary. In fact, in such cases, the Finite Elements approach approximates non-tensorisable domains with higher accuracy

with respect to finite differences approaches. In addition, to make the comparison fair, at every iteration of the loop, we compute and print the number of nodes used with the finite differences approach of `deSolve`, and the number of degrees of freedom in the mesh used by `femR`. Again, to make the comparison meaningful, in the test we set the finite elements order to be $r = 1$.

We start by presenting the time performance of our method. Figure 5.8 shows the average computational times of our numerical experiments, which we repeated several times to reduce the influence of the machine activity. As it can be noticed, the `C++` code we implemented achieves better results in terms of performance.

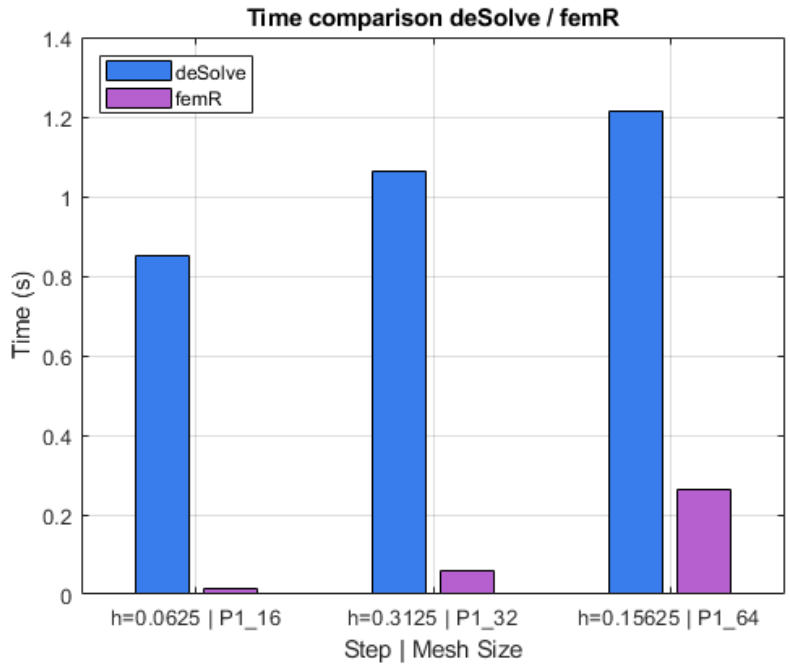

Figure 5.8: Performance of `femR` code versus `deSolve`.

The time result showed has to be interpreted together with the data of the nodes and degrees of freedom employed by the two methods, which we show in table 5.1. As the

	h=0.0625 | P1_16	**h=0.3125 | P1_32**	**h=0.15625 | P1_64**
deSolve `nodes`	256	1024	4096
femR `ndofs`	289	1089	4225

Table 5.1: Number of nodes or degrees of freedom employed by `deSolve` and `femR` on different meshes / discretization steps.

number of nodes and degrees of freedom is almost the same for every test case, we reckon this comparison fair.

After investigating the numerical performance, we show in figure 5.9 the plots of the solution obtained with both approaches, to be compared with the exact solution, also reported in figure 5.9. Moreover, we report in table 5.2 the numerical errors.

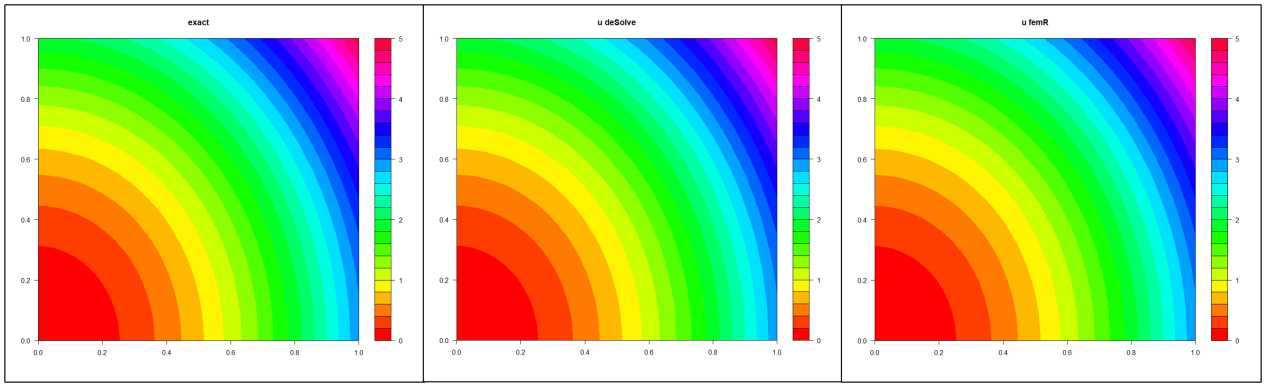

Figure 5.9: On the left, the exact solution plotted from the `R` interface. In the middle, the solution computed with the `deSolve` package. On the right, the solution computed with the `femR` library.

	h=0.0625 | P1_16	**h=0.3125 | P1_32**	**h=0.15625 | P1_64**
deSolve `error`	$2.656\,980 \times 10^{-3}$	$6.637\,046 \times 10^{-4}$	$1.658\,960 \times 10^{-4}$
femR `error`	$8.746\,182 \times 10^{-4}$	$2.289\,468 \times 10^{-4}$	$5.736\,060 \times 10^{-5}$

Table 5.2: L^2-errors of the numerical results obtained while solving problem 5.1 using `deSolve` and `femR` with linear finite elements.

We can notice that, with a competitive time performance, the femR library achieves slightly better results in terms of L^2- error.

Finally, we try to sovle once again problem 5.1 using the `femR` library, exploiting quadratic Finite Elements. While performing this test in `R`, we don't have any reference to compare the `femR` library with, therefore we limit to present or results. The plots are similar to those in figure 5.9, as the errors were already small enough to produce visually accurate solutions. We report in table 5.3 the errors obtained for this test.

	h=0.0625 | P1_16	**h=0.3125 | P1_32**	**h=0.15625 | P1_64**
femR `error`	$6.452\,227 \times 10^{-15}$	$3.183\,082 \times 10^{-14}$	$6.977\,123 \times 10^{-14}$

Table 5.3: L^2-errors of the numerical results obtained while solving problem 5.1 using `deSolve` and `femR` with quadratic finite elements.

As we can see, the errors dropped by many orders of magnitude and are close to machine precision.

6 | Conclusions

Our project aims to enhance the capabilities of the `femR` library for solving Diffusion Nonlinear-Reaction Partial Differential Equations. The expanded capabilities open up possibilities for addressing a broader range of statistical models applicable to both 2D and 3D domains.

Working on this project, we had to deal with two aspects: the theoretical background we have used to solve the considered problem and the practical implementation. Initially, we examined numerical methods designed to address nonlinear systems. Within these methods, we identified two distinct categories: one tailored to deal directly with the PDE and the other that addresses the system of nonlinear equations arising from spatial discretization. Then, we proceeded to implement a nonlinear operator and three distinct solvers employing the Fixed Point method, the Quasi-Newton method applied at the PDE level, and the Broyden algorithm, respectively. In order to accomplish this, a thorough investigation of the `femR` library was essential to understand how to extend it to our purpose.

Analyzing the results obtained with this implementation, it is clear that the most efficient solver is the one that uses the Quasi-Newton method applied to the PDE level. Additionally, good outcomes were achived with the Fixed Point solver. Using the Broyden solver, instead, we loose efficiency but its strength relies in its versatility as a general approach. Moreover, we have seen that our results are consistent both with the theory and with the results obtained with the `FreeFem++` library, noticing also that our computational time reached the same order of magnitude of that of `FreeFem++`. Lastly, comparing our `R` implementation with `deSolve`, a tool in `R` deisnged for solving Partial Differential Equations, we can see that we have achieved a significantly simpler interface obtaining lower errors in the tests.

Future developments may involve the creation of nonlinearities for either the diffusion or the transport term. This can be achieved without difficulties following our implementation for the Nonlinear-Reaction case, starting from the base class we created for the nonlinearities. Moreover, being able to manage nonlinearities, allows to insert them

into statistical models with a PDE regularization. This last step may not be immediate, especially from the theoretical viewpoint.

7 | Installation of femR and Instructions

7.1. Installation

The code created for this project is available in the GitHub repository through the following link: `https://github.com/ema-tambu/fdaPDE-core/tree/develop`. This repository is a fork of the `fdaPDE/fdaPDE-core` repository, which contains the `C++`, header-only, core library system for the `fdaPDE` project. Since we are considering a header-only library, no installation is required.

To compile code included in this library the user needs:

- A `C++17` compliant compiler. Supported versions are:
  - Linux: gcc 11 (or higher), clang 13 (or higher)
  - macOS: apple-clang (the XCode version of clang).
- The Eigen linear algebra library, version 3.3.9.

7.2. Run tests from C++

In the `fdaPDE-core` library there's a test suite contained in the `test/` folder. To run all of these tests, make sure that Google Test is installed, then proceed using the following bash commands:

```
cd fdaPDE-core/test
cmake CMakeLists.txt
make
./fdapde_test
```

The tests we have developed for the nonlinear operator, the solvers and Broyden algorithm con be found, respectively, in the files `test/src/fem_operators_test.cpp`, `test/src/fem_pde_test.cpp` and `test/src/optimization_test.cpp`

7.3. R wrapper

femR is the R wrapper to the fdaPDE Finite Element solver for Partial Differential Equations.

In order to use it, make sure to have the following dependencies installed on your system:

- a C++17 compliant compiler,
- the Rcpp, RcppEigen, R6, sf, RTriangle packages.

Then, to install the latest stable version of femR, you can use the devtools package. From the R console, execute

```
devtools::install_github("ema-tambu/pacs-project-femR", ref="develop")
```

The test we developed to compare femR package and deSolve package in solving a non-linear PDE can be executed from a terminal as follows:

```
Rscript tests/non_linear_reaction_deSolve.R
```

Bibliography

[1] H. Brezis. *Functional Analysis, Sobolev Spaces and Partial Differential Equations*. Universitext. Springer New York, 2010. ISBN 9780387709130. URL https://books.google.it/books?id=GAA2XqOIIGoC.

[2] C. G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 1965. ISSN 00255718, 10886842. URL http://www.jstor.org/stable/2003941.

[3] P. Chaggar, J. Vogel, A. P. Binette, T. B. Thompson, O. Strandberg, N. Mattsson-Carlgren, L. Karlsson, E. Stomrud, S. Jbabdi, S. Magon, G. Klein, the Alzheimer's Disease Neuroimaging Initiative, O. Hansson, and A. Goriely. Personalised regional modelling predicts tau progression in the human brain. *bioRxiv*, 2023. doi: 10.1101/2023.09.28.559911. URL https://www.biorxiv.org/content/early/2023/09/29/2023.09.28.559911.

[4] J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, volume 16 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, USA, 1996.

[5] P. Draper. Statistical methods and models for complex data. Department of Statistical Science, Padova., 2022.

[6] L. Evans. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 2010. ISBN 9780821849743. URL https://books.google.it/books?id=Xnu0o_EJrCQC.

[7] fdaPDE contributors. The fdapde core library. GitHub repository, 2024. URL https://github.com/ema-tambu/fdaPDE-core.

[8] fdaPDE contributors. femr: An r package for solving partial differential equations using the finite element method. GitHub repository, 2024. URL https://github.com/ema-tambu/pacs-project-femR.

[9] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012. ISSN 1570-2820. URL https://freefem.org/.

[10] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Frontiers in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia, Jan. 1995. doi: 10.1137/1.9781611970944. URL http://www.siam.org/books/textbooks/fr16_book.pdf.

[11] D. Li, J. Zeng, and S. Zhou. Convergence of broyden-like matrix. *Applied Mathematics Letters*. doi: https://doi.org/10.1016/S0893-9659(98)00076-7. URL https://www.sciencedirect.com/science/article/pii/S0893965998000767.

[12] J. T. Nardini and D. M. Bortz. Investigation of a structured fisher's equation with applications in biochemistry. *SIAM Journal on Applied Mathematics*, 2018. doi: 10.1137/16M1108546. URL https://doi.org/10.1137/16M1108546.

[13] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, 2e edition, 2006.

[14] A. Palummo. Functional principal component analysis for spacetime data over complex domains and restructuring of the fdapde library. Master's thesis, Politecnico di Milano, 2023.

[15] H. K. Pathak. *Differential Calculus in Banach Spaces*. Springer Singapore, 2018. ISBN 978-981-10-8866-7. doi: 10.1007/978-981-10-8866-7_3. URL https://doi.org/10.1007/978-981-10-8866-7_3.

[16] J. Reddy. *An Introduction to Nonlinear Finite Element Analysis*. OUP Oxford, 2004. ISBN 9780198525295. URL https://books.google.it/books?id=gV2EN71IjwgC.

[17] S. Salsa. *Partial Differential Equations in Action: From Modelling to Theory*. UNITEXT. Springer Cham, 2016. ISBN 978-3-319-31238-5. doi: 10.1007/978-3-319-31238-5. URL https://doi.org/10.1007/978-3-319-31238-5.

[18] S. Salsa and G. Verzini. *Partial Differential Equations in Action: From Modelling to Theory*. UNITEXT. Springer Cham, 2022. ISBN 978-3-031-21852-1. doi: 10.1007/978-3-031-21853-8. URL https://doi.org/10.1007/978-3-031-21853-8.

[19] L. M. Sangalli. Spatial regression with partial differential equation regularisation. *International Statistical Review*, December 2021. doi: 10.1111/insr.12444. URL https://ideas.repec.org/a/bla/istatr/v89y2021i3p505-531.html.

[20] J. R. Senning. Computing and estimating the rate of convergence, 2007. URL https://www.math-cs.gordon.edu/courses/ma342/handouts/rate.pdf.

[21] K. Soetaert and F. Meysman. Reactive transport in aquatic ecosystems: Rapid model prototyping in the open source software r. *Environmental Modelling Software*, 2012. doi: https://doi.org/10.1016/j.envsoft.2011.08.011. URL https://www.sciencedirect.com/science/article/pii/S1364815211001940.

[22] K. Soetaert, T. Petzoldt, and R. W. Setzer. Solving differential equations in r: Package desolve. *Journal of Statistical Software*, 2010. doi: 10.18637/jss.v033.i09. URL https://www.jstatsoft.org/index.php/jss/article/view/v033i09.

[23] Wikipedia contributors. Fisher's equation, 2024. URL https://en.wikipedia.org/wiki/Fisher%27s_equation. Accessed on 1 February 2024.

[24] Wikipedia contributors. Gâteaux derivative, 2024. URL https://en.wikipedia.org/wiki/Gateaux_derivative.

[25] L. Zhang. A derivative-free conjugate residual method using secant condition for general large-scale nonlinear equations. *Numer. Algorithms*, page 1277–1293, 2020. doi: 10.1007/s11075-019-00725-7. URL https://doi.org/10.1007/s11075-019-00725-7.

[26] W. Zhou and L. Zhang. A modified broyden-like quasi-newton method for nonlinear equations. *Journal of Computational and Applied Mathematics*, 07 2020. doi: 10.1016/j.cam.2020.112744.

A | Appendix A

We report in this appendix the full FreeFem++ scripts that we implemented to test our results.

We implemented the Fixedpoint method in FreeFem++ as shown in the following listing:

```
int n = 16;
mesh Th = square(n, n); //flags = 1);
plot(Th);
fespace Vh(Th, P2);
Vh fh, vh, f0h, incr, Err, fexh;
// 1. exact solution (useful for testing)
func fex = 3*x*x + 2*y*y;
real nu = 1;
// define forcing term
func u = -9*x^4 - 12*x^2*y^2 + 3*x^2 - 4*y^4 + 2*y^2 - 10*nu;
fexh = fex; // Interpolate fex into Vh
int Ndofs = Vh.ndof;
cout << "Ndofs = " << Ndofs << endl;
// 3. define problem
problem fixedpoint(fh, vh, solver = LU, eps = 1.e-10, init = 0) =
    int2d(Th)(
        nu * (dx(fh)*dx(vh) + dy(fh)*dy(vh)) //Laplacian
        + (1.0 - f0h) * fh * vh  //NonLinear
    )
    - int2d(Th)(u * vh)
    + on(1, 2, 3, 4, fh = fex);
func real increment(){
  // store increments in incr variables
  incr[] = fh[];
  incr[] -= f0h[];
  real result = int2d(Th)(incr^2);
  return sqrt(result);
}
// Compute mass matrix for weighted error.
varf mass(p, q) = int2d(Th)(p * q);
matrix M = mass(Vh, Vh);
real[int] prod(Ndofs);
// function to compute the error
func real error(){
  // store increments in incr variables
  Err[] = fexh[];
  Err[] -= fh[];
  for (int i = 0; i < Ndofs; ++i)
    prod[i] = Err[][i]*Err[][i];
```

```
  real[int] temp(Ndofs);
  temp = M*prod;
  real err2 = 0;
  for (int i = 0; i< Ndofs; ++i)
      err2 += temp[i];
  return err2;
}
// loop control
int nbiter = 20;
real eps = 1e-7;
int iter;
real incrErr = eps + 1; // at least one iteration
real err = eps + 1;
[fh] = [0.0];
fixedpoint;
err = error();
cout << "\t Newton iteration 0, error = " << err << endl;
[f0h] = [fh];
// for (iter = 0; iter < nbiter && incrErr > eps; ++iter) {
for (iter = 0; iter < nbiter && err > eps; ++iter) {
    fixedpoint; // perform a fixed point iteration
    // incrErr = increment(); // update incr for the stopping criterion
    err = error(); // update error for the stopping criterion
    [f0h] = [fh]; // update old solution
    // cout << "\t Fixedpoint iteration " << iter+1 << ", incr = " << incrErr << endl;
    cout << "\t Fixedpoint iteration " << iter+1 << ", error = " << err << endl;
}
cout << "# iter: " << iter+1 << endl;
plot([fh], value = 1, fill = 1, wait = 1, cmm = "f_computed");
plot([fexh], value = 1, fill = 1, wait = 1, cmm = "Exact Solution");
plot(Th, wait = 1, value =1, cmm = "Discretization unit_square_16");
cout << "\n\t saving results... " << endl;
ofstream file("freefem_P1_fixedpoint.txt");
for(int i=0; i<fh[].n; i++){
    file << fh[][i] << endl;
}
```

We report here the Newton method implementation in `FreeFem++` as well:

```
int n = 16;
mesh Th = square(n, n); //flags = 1);
plot(Th);
fespace Vh(Th, P2);
Vh fh, vh, f0h, incr, Err, fexh;
// 1. exact solution (useful for testing)
func fex = 3*x*x + 2*y*y;
real nu = 1;
// define forcing term
func u = -9*x^4 - 12*x^2*y^2 + 3*x^2 - 4*y^4 + 2*y^2 - 10*nu;
fexh = fex; // Interpolate fex into Vh
int Ndofs = Vh.ndof;
cout << "Ndofs = " << Ndofs << endl;
// 3. define problems
problem fixedpoint(fh, vh, solver = LU, eps = 1.e-10, init = 0) =
    int2d(Th)(
        nu * (dx(fh)*dx(vh) + dy(fh)*dy(vh)) //Laplacian
```

```
        + (1.0 - f0h) * fh * vh  //NonLinear
    )
    - int2d(Th)(u * vh)
    + on(1, 2, 3, 4, fh = fex);
problem newton(fh, vh, solver = LU, eps = 1.e-10) =
        nu * (dx(fh)*dx(vh) + dy(fh)*dy(vh)) //Laplacian
        + (1.0 - f0h) * fh * vh  // NonLinear h(x, f_k) f_{k+1}
        + (-1) * f0h * fh * vh   // NonLinearityPrime h'(x, f_k) * f_k * f_{k+1} - in our
             case h' = -1 !!
    )
    - int2d(Th)(
        u * vh
        + (-1) * f0h * f0h * vh    // h'(x, f_k)f_k f_k
    )
    + on(1, 2, 3, 4, fh = fex);
// Compute mass matrix for weighted error.
varf mass(p, q) = int2d(Th)(p * q);
matrix M = mass(Vh, Vh);
real[int] prod(Ndofs);
func real increment(){
  // store increments in incr variables
  incr[] = fh[];
  incr[] -= f0h[];
  real result = 0;
  for (int i = 0; i < Ndofs; ++i)
    result += incr[][i]*incr[][i];
  return sqrt(result);
}
func real error(){
  // store increments in incr variables
  Err[] = fexh[];
  Err[] -= fh[];
  for (int i = 0; i < Ndofs; ++i)
    prod[i] = Err[][i]*Err[][i];
  real[int] temp(Ndofs);
  temp = M*prod;
  real err2 = 0;
  for (int i = 0; i< Ndofs; ++i)
      err2 += temp[i];
  return err2;
}
// loop control
int nbiter = 20;
real eps = 1e-7;
int iter;
real incrErr = eps + 1; // at least one iteration
real err = eps + 1;
[fh] = [0.0];
fixedpoint;     // solve associated linear system
err = error();
cout << "\t␣Newton␣iteration␣0,␣error␣=␣" << err << endl;
[f0h] = [fh];
// for (iter = 0; iter < nbiter && incrErr > eps; ++iter) {
for (iter = 0; iter < nbiter && err > eps; ++iter) {
    newton; // perform a newton iteration
    // incrErr = increment(); // update incr for the stopping criterion
```

```
    err = error(); // update error for the stopping criterion
    [f0h] = [fh]; // update old solution
    cout << "\t Newton iteration " << iter+1 << ", incr = " << incrErr << endl;
    cout << "\t Newton iteration " << iter+1 << ", error = " << err << endl;
}
cout << "# iter: " << iter + 1<< endl;
plot([fh], value=1, fill = 1, wait = 1, cmm="f_computed");
plot([fexh], value=1, fill=1, wait=1, cmm="Exact Solution");
cout << "\n\t saving results... " << endl;
ofstream file("freefem_P1_newton.txt");
for(int i=0; i<fh[].n; i++){
    file << fh[][i] << endl;
}
```

The results obtained with the `FreeFem++` implementation are numerically the same obtained with our implementation in the `fdaPDE/core` module, as the outputs differ with an order of magnitude of 1×10^{-30} at every iteration.