UNIVERSITY OF TRIESTE

Department of Engineering and Architecture

Master's Degree in
Computer & Electronic Engineering

Shape optimization with discrete adjoint method applied to RBF-FD meshless method

August 14, 2024

Candidate
Kevin Marzio

Supervisor
Prof. Andrea De Lorenzo

Co-supervisors
Dr. Mauro Munerato
Dr. Riccardo Zamolo
Dr. Davide Miotti

A.Y. 2023/2024

Lorem ipsum
dolor sit amet

- Cicero

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Contents

Introduction

Spiegazione/storia mesh-based.
Perché siamo arrivati ai meshless e differenze in generale
Perché diff. automatica e aggiunto nel meshless

Chapter 1

Meshless methods

Meshless or Meshfree Methods (MMs) were developed to overcome the drawbacks of traditional mesh-based methods for the solution of Partial Differential Equations (PDE), where their true advantage is that "the approximation of unknowns in the PDE is constructed based on scattered points without mesh connectivity" [1]. The conceptual difference between Mesh Based and Meshless methods can be visualized on figure 1.1: the former patch the domain with some geometrical shapes while the latter only distributes nodes across the domain.

They appeared for the first time in 1977 with the Smooth Particle Hydrodynamics (SPH) method [2], initially used to modeling astrophysical phenomena such as exploding stars and dust clouds was later applied in solid mechanics to overcome limitations of mesh-based methods [3]. To improve accuracy, tensile instability and spatial instability of SPH many more modern MMs have been developed: the introduction of Reproducing Kernel Particle Method (RKPM) [4] is a prime example of enhanced consistency and stability. Generalized Finite Difference (GFD) methods is another branch of numerical methods for solving PDEs that do not rely on a grid structure and many modern MMs originate from the employment of this approximation for solving PDEs.

The typical use case for MMs is the construction of an approximating field u^h for the sought solution $u\colon \Omega \subset \mathbb{R}^d \to \mathbb{R}$ of the following boundary value problem:

$$\begin{cases} \mathcal{L}u(\boldsymbol{x}) = f(\boldsymbol{x}) & \text{in } \Omega \\ \mathcal{B}u(\boldsymbol{x}) = g(\boldsymbol{x}) & \text{on } \partial\Omega \end{cases} \tag{1.0.1}$$

where $\mathcal{L}$ is a generic linear differential operator and $\mathcal{B}$ is a different linear operator used to enforce some Boundary Condition (BC) that does not necessarily involves partial derivatives; f and g are known functions. Usually in the Computational

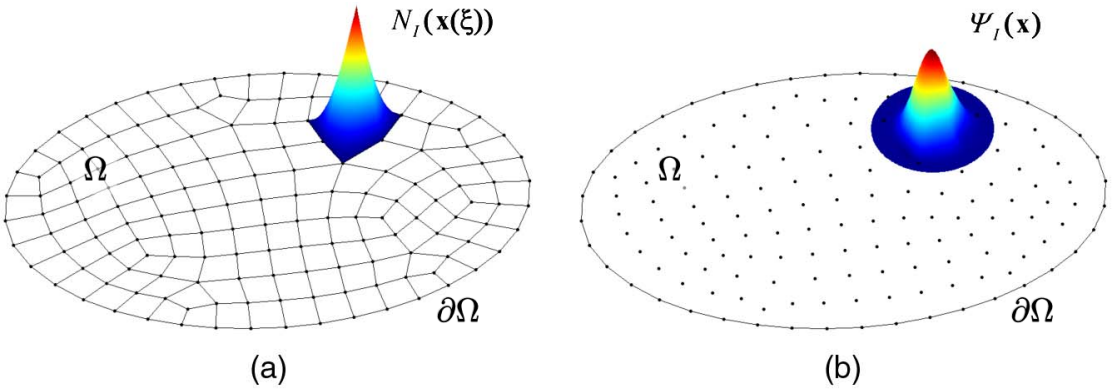


Figure 1.1: (a) Example of reconstruction of a PDE solution, via mesh based method, for a single finite element domain; (b) approximation of the same function obtained through meshless approach. Figure taken from [1]

Fluid Mechanics (CFD) world the encountered BC are:

$$\text{Dirichlet BC:} \qquad u = g \tag{1.0.2}$$

$$\text{Neumann BC:} \qquad \frac{\partial u}{\partial \boldsymbol{n}} = g \tag{1.0.3}$$

$$\text{Robin BC:} \qquad au + b\frac{\partial u}{\partial \boldsymbol{n}} = g \tag{1.0.4}$$

where $\frac{\partial u}{\partial \boldsymbol{n}}$ indicate the normal derivative of u whereas a and b are known functions. It can also be noticed that Dirichlet and Neumann BCs are special cases of Robin BC respectively when $b(\boldsymbol{x}) = 0$ and $a(\boldsymbol{x}) = 0$.

Regardless on the Meshfree approach, the following approximation for any solution u can be written:

$$u^h(\boldsymbol{x}) = \sum_{k=1}^{N} \alpha_k B_k(\boldsymbol{x}) \tag{1.0.5}$$

where $B_k \colon \Omega \to \mathbb{R}$ are suitable basis functions and α_k are the expansions coefficients that must be determined. Different choices for the basis functions B_k leads to different formulations and in literature can be found several of these, some examples are: RKPM [4], moving least square (MLS) [5], radial basis function (RBF) [6, 7] and partition of unity (PU) [8]. Furthermore solving the PDE in (1.0.1) with the approximated solution u^h, in general, yields to a non-zero error function ϵ^h given by:

$$\epsilon^h(\boldsymbol{x}) = \mathcal{L}u^h(\boldsymbol{x}) - f(\boldsymbol{x}) \tag{1.0.6}$$

Once the general form of the approximated solution in (1.0.5) has been properly defined, it can be employed for discretizing the PDE, reported in (1.0.1), over a set

of N generated nodes $\{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_N \}$ distributed in the physical domain $\Omega \cup \partial\Omega$. The Weighted Residual Method is used to do so: a set of test functions $\{ \Gamma_1, \dots, \Gamma_N \}$ orthogonal to ϵ^h are used to integrate the error to zero:

$$\begin{aligned} \int_\Omega \Gamma_i \epsilon^h \, d\Omega &= \int_\Omega \Gamma_i (\mathcal{L} u^h(\boldsymbol{x}) - f(\boldsymbol{x})) \, d\Omega \\ &= \int_\Omega \Gamma_i \Big[\mathcal{L} \Big(\sum_{k=1}^{N} \alpha_k B_k(\boldsymbol{x}) \Big) - f(\boldsymbol{x}) \Big] \, d\Omega = 0 \end{aligned} \tag{1.0.7}$$

where $i = 1, \dots, N$. From the choice of functions Γ_i the following two formulations are obtained [1]:

Galerkin Meshless Methods that find a weak solution for the PDE by using as test functions the basis functions B_k. These formulations require domain integration and special techniques to enforce boundary conditions;

Collocation Meshless Methods that find a strong solution for the PDE by using Dirac delta functions centered at the discretization nodes as test functions. Basically they enforce equaitons (1.0.1) on a finite set of nodes, and by doing so, they do not require any domain integration.

Finally, once constraints (1.0.7) are enforced, the solution of problem (1.0.1), i.e. the values $\{ u^h(\boldsymbol{x}_1), \dots, u^h(\boldsymbol{x}_N) \}$, can be found solving the linear system resulting from PDE's discretization. We remark that the system obtained is not linear in general, but it is in this case since the PDE is linear. Nevertheless, also non-linear PDEs can be reduced to the aforementioned case thanks to a proper linearization.

Chapter 2

RBF-FD method

In this chapter we explain in more details the Meshless Method (MM) used within this work: the Radial Basis Function generated Finite Differences (RBF-FD) method. To do so we show how it can be used to solve the general boundary value problem defined in chapter 1. From now on, in order to be able to discretize the PDE, we consider to have a disposal a set of N distinct nodes, $\mathcal{X}$, defined as follow:

$$\mathcal{X} := \{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_N \mid \boldsymbol{x}_i \in \Omega \cup \partial\Omega,\ i = 1, \dots, N \} \tag{2.0.1}$$

where $\Omega \cup \partial\Omega \subset \mathbb{R}^d$ is the physical domain of the problem, Ω and $\partial\Omega$ indicate respectively its open subset and boundary, and $d \in \mathbb{N}$ is its dimension.

2.1 Scattered Data Interpolation

Every MM, as we have already seen, on its core, is just a way to approximate the solution of a PDE and RBF-FD method is no exception; the tool used to do so is called *scattered data interpolation*. In this section we first see what scattered data interpolation is and then how it is related to PDEs solution. In the second part of the explanation we see how it is applied in general by MMs so to avoid to becoming fixated on a single implementation and losing generality; moreover this approach is still useful since it establishes all the steps that are also followed by RBF-FD. To avoid from the very beginning any kind of ambiguity we underline that scattered data interpolation is inherently connected to data fitting and not to PDEs approximation, consequently, it finds many other applications beyond the realm of MMs. Examples of its application on the field of image processing can be found in [14].

In general the interpolation problem has the following, simple, formulation. Given:

- a finite set of nodes, $\mathcal{X} \subset \mathbb{R}^d$, which can be the one reported in (2.0.1) and;

- a set of known real values $u(\boldsymbol{x}_1), \dots, u(\boldsymbol{x}_N)$, which may be obtained from a function

we want to find a continuous function $u^h : \Omega \cup \partial\Omega \subset \mathbb{R}^d \to \mathbb{R}$ that satisfy:

$$u^h(\boldsymbol{x_i}) = u(\boldsymbol{x}_i) \qquad \forall \boldsymbol{x}_i \in \mathcal{X} \tag{2.1.1}$$

If the locations, the nodes in $\mathcal{X}$ where the measurements $u(\boldsymbol{x}_1), \dots, u(\boldsymbol{x}_N)$ are taken, are placed on a uniform or regular grid we talk about interpolation, otherwise the process above is called *scattered* data interpolation. Here the main idea is to find a function u^h which is a "good" fit to the given data, where with "good" we mean a function that exactly match the given measurements at the corresponding locations.

MMs also aim to provide an approximation for an unknown function defined as a linear combination of a set of basis functions as reported in equation (1.0.5); we report here for clarity its definition:

$$u^h(x) = \sum_{k=1}^{N} \alpha_k B_k(\boldsymbol{x}) \tag{2.1.2}$$

and we remark that coefficients α_k are unknown. Here is where the theory of scattered data interpolation is applied: it tells us that we are able to find the numerical values for $\alpha_1, \dots \alpha_N$ if we impose a number of conditions equal to the number of coefficients that we are looking for. These conditions must have a form like that shown in equation (2.1.1). Therefore if we replace the generic meshless approximation within each of the N interpolation conditions we can write the following linear system:

$$\underbrace{\begin{bmatrix} B_1(\boldsymbol{x}_1) & \dots & B_N(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ B_1(\boldsymbol{x}_N) & \dots & B_N(\boldsymbol{x}_N) \end{bmatrix}}_{\boldsymbol{B}} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} u(\boldsymbol{x}_1) \\ \vdots \\ u(\boldsymbol{x}_N) \end{bmatrix} \tag{2.1.3}$$

that, once solved, provides us the desired coefficients that uniquely define u^h given the set of basis functions $\{ B_1, \dots, B_N \}$. We would like to stress the fact that, however, the right hand side vector is made up of values of the unknown exact solution of problem (1.0.1).

During the solution of problem (1.0.1), via Collocation Methods, the following constraints arise instead:

$$\begin{aligned} \mathcal{L}u^h(\boldsymbol{x}_j) &= f(\boldsymbol{x}_j) \quad \text{if } \boldsymbol{x}_j \in \Omega \\ u^h(\boldsymbol{x}_j) &= g(\boldsymbol{x}_j) \quad \text{if } \boldsymbol{x}_j \in \partial\Omega \end{aligned} \tag{2.1.4}$$

and if we arrange the nodes such that the first N_I nodes belongs to Ω and the last N_B to $\partial\Omega$, we can find the values of the approximated solution at the points in $\mathcal{X}$ by solving:

$$\begin{bmatrix} c_{1,1} & \dots & c_{1,N_I} \\ \vdots & \ddots & \vdots \\ c_{N_I,1} & \dots & c_{N_I,N_I} \end{bmatrix} \begin{bmatrix} u^h(\boldsymbol{x}_1) \\ \vdots \\ u^h(\boldsymbol{x}_{N_I}) \end{bmatrix} = \boldsymbol{f} - \begin{bmatrix} c_{1,N_I+1} & \dots & c_{1,N_B} \\ \vdots & \ddots & \vdots \\ c_{N_I,N_I+1} & \dots & c_{N_I,N_B} \end{bmatrix} \boldsymbol{g} \tag{2.1.5}$$

where $\boldsymbol{f} = [f(\boldsymbol{x}_1) \dots f(\boldsymbol{x}_{N_I})]^T$ and $\boldsymbol{g} = [g(\boldsymbol{x}_{N_I+1}) \dots g(\boldsymbol{x}_{N_B})]^T$, and the coefficient matrix $\boldsymbol{C}$ is found as the solution of:

$$\begin{bmatrix} c_{1,1} & \dots & c_{N_I,1} \\ \vdots & \ddots & \vdots \\ c_{1,N} & \dots & c_{N_I,N} \end{bmatrix} = \boldsymbol{B}^{-T} \begin{bmatrix} \mathcal{L}B_1(\boldsymbol{x}_1) & \dots & \mathcal{L}B_1(\boldsymbol{x}_{N_I}) \\ \vdots & \ddots & \vdots \\ \mathcal{L}B_N(\boldsymbol{x}_1) & \dots & \mathcal{L}B_N(\boldsymbol{x}_{N_I}) \end{bmatrix} \tag{2.1.6}$$

Analyzing each row $\boldsymbol{c}_i = [c_{i,1}, \dots c_{i,N}]$ of matrix $\boldsymbol{C}$ we can notice that are computed solving the following linear systems:

$$\boldsymbol{B}^T c_i = \begin{bmatrix} \mathcal{L}B_1(\boldsymbol{x_i}) \\ \vdots \\ \mathcal{L}B_N(\boldsymbol{x_i}) \end{bmatrix} \qquad i = 1, \dots, N_I \tag{2.1.7}$$

which are closely related to the ones obtained from scattered data interpolation reported in (2.1.3) due to the presence of the same matrix $\boldsymbol{B}$. We conclude by commenting that equation (2.1.6) with matrix $\boldsymbol{B}$ defined as in (2.1.3) holds true only in case of Dirichlet boundary conditions, otherwise $\boldsymbol{B}$ would take on a different form.

2.2 The Mairhuber-Curtis theorem

From the previous discussion, in particular from equation (2.1.6), it can be noticed that matrix $\boldsymbol{B}$ has to be non singular in order to be able to solve the linear system associated to the boundary value problem, and this must hold for each node placement $\mathcal{X}$ (to be read as every possible discretization of the problem domain) as long nodes are distinct. This property of $\boldsymbol{B}$ turns out to be dependent on the choice of the particular set of basis functions: for example if we assume $B_k(\boldsymbol{x}) \in \Pi_P^d$ and $\{ B_1(\boldsymbol{x}), \dots, B_N(\boldsymbol{x}) \}$ to be a polynomial basis of the space Π_P^d of polynomials of degree at most P in $\mathbb{R}^d$, then we are not able to guarantee that $\boldsymbol{B}$ is invertible for $d > 1$.

This issue is explained in more detail by the Mairhuber-Curtis theorem [15]; when dealing with the multidimensional case it is possible to continuously move

two nodes along a closed path P, that does not interfere with any other node in $\mathcal{X}$, such that they end up by interchanging their original positions without one crossing the path of the other. In the event that these two are the only nodes of $\mathcal{X}$ that are moved, $\boldsymbol{B}$ ends up with two rows exchanged leading to a change in the sign of its determinant, and, since the determinant is a continuous function, this means that there is a moment when the latter vanishes making the matrix singular.

The inconvenience arises from the fact that the set of basis functions is independent from the node position and could be solved by simply choosing a basis that is function of nodes position. By doing so we no more fall in the case of the Mairhuber-Curtis theorem since whenever we move nodes also the base itself changes and if two nodes switches their positions not only their respective rows in $\boldsymbol{B}$ are switched, but also their columns, forcing the determinant not to change in sign.

2.3 Radial Basis Functions

Up to now, when talking about the approximated solution of the PDE, u^h, we have not specified the type of basis functions which define it. However these must be done in order to be able to find the coefficients α_k in equation (2.1.2) and thus its numerical values; furthermore it would be appropriate to select a set of functions that allow the avoidance of the aforementioned Mairhuber-Curtis' theorem case. In this section we will define the ones used by the RBF-FD method: the Radial Basis Functions (RBFs).

RBFs are defined as:

$$\Phi(\boldsymbol{x}, \boldsymbol{x}_k) = \varphi(\|\boldsymbol{x} - \boldsymbol{x}_k\|_2) \tag{2.3.1}$$

where $\boldsymbol{x}_k \in \mathbb{R}^d$ is a given and known point , $\|\cdot\|_2$ is the euclidean distance and $\varphi\colon \mathbb{R} \to \mathbb{R}$, named *basic* function, is a (univariate) function which takes the radius $r_k = \|\boldsymbol{x} - \boldsymbol{x}_k\|_2$ as input and it is used as generator for all the (multivariate) *basis* functions associated to different $\boldsymbol{x}_k$.

Sometimes the notations $\Phi(\boldsymbol{x} - \boldsymbol{x}_k)$ or $\Phi_k(\boldsymbol{x})$ are also used instead of $\Phi(\cdot, \boldsymbol{x}_k)$ where "$\cdot$" act as a placeholder for the corresponding argument meaning that varies freely. However the 2-arguments notation makes clear that $\Phi(\cdot, \cdot)$ is a real valued function defined on the space given by the Cartesian product $\Omega \times \Omega$. This type of functions, in accordance with the definition given in [16], are called *kernels*. In general at different basic functions φ are associated different kernels; some of these are reported in table 2.1.

To further clarify the RBFs name we can notice that they are called:

Radial since the value of each $\Phi(\cdot, \boldsymbol{x}_k)$ at each point $\boldsymbol{x}$ depends only on the

Table 2.1: Examples of basic functions where r is a real number greater than or equal to zero, and ϵ, called shape factor, is a suitable parameter

Name	$\varphi(r)$
Multiquadratic	$\sqrt{1+(\epsilon r)^2}$
Inverse multiquadratic	$\left(\sqrt{1+(\epsilon r)^2}\right)^{-1}$
Thin plate splines	$r^{2l}\log l,\ l \in \mathbb{N}$
Gaussian	$e^{-(\epsilon r)^2}$
Polyharmonics	$r^{2l+1},\ l \in \mathbb{N}$

distance between that point and $\boldsymbol{x}_k$ through $\|\cdot\|_2$. Basically they satisfy radial symmetry, i.e. $\Phi(\boldsymbol{x}_i, \boldsymbol{x}_j) = \Phi(\boldsymbol{x}_j, \boldsymbol{x}_i)$ for any $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathbb{R}^d$;

Basis since in case of a set of N distinct nodes in $\mathbb{R}^d$, as can be $\mathcal{X}$ given in (2.0.1), the set of radial functions $\Phi_k(\boldsymbol{x})$ with $k = 1, \dots, N$ associated to each point of the set, form a basis for the space of functions:

$$F_\Phi := \left\{ \sum_{k=1}^{N} \alpha_k \Phi_k(\boldsymbol{x}), \quad \alpha_k \in \mathbb{R},\ \boldsymbol{x}_k \in \mathcal{X} \right\}$$

In case of RBF-FD method the implementation of scattered data interpolation and the solution of governing equation remain the same as explained in the previous section and leads to a symmetric matrix $\boldsymbol{B}$ defined as:

$$\boldsymbol{B} = \begin{bmatrix} \Phi(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & \Phi(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \ddots & \vdots \\ \Phi(\boldsymbol{x}_N, \boldsymbol{x}_1) & \dots & \Phi(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{bmatrix} \tag{2.3.2}$$

We conclude by observing that this functions are particularly convenient since they depend on nodes position through $\boldsymbol{x}_k$, thus they allow to avoid the non-invertibility of matrix $\boldsymbol{B}$ in case of singular node arrangements (Mairhuber-Curtis theorem).

2.4 Polynomial augmentation

Setting aside the issue of the invertibility of matrix $\boldsymbol{B}$ in case of particular nodes arrangement discussed in previous subsection, we should also take into account the accuracy of the interpolation that we are able to achieve, which also depends upon the type of functions that we are supposed to approximate. Indeed

RBFs approximation schemes alone are not able to exactly interpolate (i.e. with an accuracy only depending on roud-off errors) constant, linear or higher degree polynomials fields. This is an issue in different important engineering applications such as modeling of constant strain in elastic bodies and steady temperature fields in differentially heated walls [17].

To overcome this limitation, a polynomial augmentation of degree P is required, leading to the overall formulation for the RBF interpolant:

$$u^h(\boldsymbol{x}) = \sum_{j=1}^{N} \alpha_j \Phi_j(\boldsymbol{x}) + \sum_{k=1}^{M} \beta_k p_k(\boldsymbol{x}) \tag{2.4.1}$$

where $M = \binom{P+D}{D}$ is the number of polynomial basis functions with degree $P \leq D$, $\{\, p_1(\boldsymbol{x}) \dots p_M(\boldsymbol{x}) \,\}$ is a complete polynomial basis of Π_P^d and β_j are the corresponding coefficients. An example of polynomial basis for polynomials of degree $P = 1$ in $2D$ has the following $M = 3$ elements: $p_1(x, y) = 1$, $p_2(x, y) = x$, $p_3(x, y) = y$.

We must also note that using an interpolant with the introduced polynomial augmentation leads to an underdetermined system in (2.1.3). In order to obtain a square $\boldsymbol{B}$ and thus having a solvable system, the following orthogonality conditions have to be imposed:

$$\sum_{i=1}^{N} \alpha_i p_k(\boldsymbol{x}_i) = 0, \qquad k = 1, \dots, M \tag{2.4.2}$$

The coefficients of u^h, which are now composed not only by $\boldsymbol{\alpha} = [\alpha_1 \dots \alpha_N]$, but also by $\boldsymbol{\beta} = [\beta_1 \dots \beta_M]$, can then be found by solving the following system:

$$\underbrace{\begin{bmatrix} \boldsymbol{B} & \boldsymbol{P} \\ \boldsymbol{P}^T & \boldsymbol{0} \end{bmatrix}}_{\boldsymbol{M}} \begin{bmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{0} \end{bmatrix} \tag{2.4.3}$$

where:

$$\boldsymbol{P} = \begin{bmatrix} p_0(\boldsymbol{x}_1) & \dots & p_M(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ p_0(\boldsymbol{x}_N) & \dots & p_M(\boldsymbol{x}_N) \end{bmatrix} \tag{2.4.4}$$

$$\boldsymbol{u} = [u(\boldsymbol{x}_1) \dots . u(\boldsymbol{x}_N)]$$

From its formulation is easy to understand that the system above is simply the composition of the system in equation (2.1.3), in the first row, with constraints (2.4.2) written in compact form, in the second row.

In practice the addition of polynomial basis to the RBF interpolant let us perfectly fit $u(\boldsymbol{x})$ not only on collocation points where we have $u^h(\boldsymbol{x}_i) = u(\boldsymbol{x}_i)$, but

also across the rest of the domain provided that data $u(\boldsymbol{x}_1), \ldots, u(\boldsymbol{x}_N)$ come from a polynomial of total degree less than or equal to P. Nevertheless this procedure has a side effect: not all set of nodes $\mathcal{X}$ nor all basic functions φ leads to a well-posed RBF interpolation with a non singular matrix M. We will discuss this limitation in more detail in the next section.

2.5 Problem solution

At the beginning of section 2.2 we mentioned that the system in equation (2.1.6) might not be solvable in case of a combination of non-point-dependent basis functions and particular nodes arrangements, but, up to now, we did not discuss in general in which cases the interpolation problem is solvable. In this section we will address this shortcoming.

We start by recalling that, in case of pure RBF interpolant, the system that we aim to solve has the following compact form:

$$\boldsymbol{B}\boldsymbol{\alpha} = \boldsymbol{u} \tag{2.5.1}$$

where $\boldsymbol{B}$ is a symmetric matrix. A positive definite $\boldsymbol{B}$ would be sufficient in order to solve the above system and this last property depends on the choice of the basic function φ used to define the RBFs. By definition only *strictly* positive definite [18] φ are associated to a positive definite $\boldsymbol{B}$ and, this, restrict our choices: of those shown in table 2.1 only Inverse Multiquadratic and Gaussian functions satisfy this requirement. The same attributes of φ are also inherited by the associated RBFs Φ_k.

However in previous section we have seen that, beyond the solvability issue, a polynomial augmentation of degree P is beneficial for the accuracy of the interpolant u^h. This means that the system that we have to solve, in general, is no more in the form shown in equation (2.5.1), but rather in the following:

$$\boldsymbol{M} \begin{bmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{0} \end{bmatrix} \tag{2.5.2}$$

In this case the matrix we seek to be positive definite would be $\boldsymbol{M}$that we recall being defined as:

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{B} & \boldsymbol{P} \\ \boldsymbol{P}^T & \boldsymbol{0} \end{bmatrix} \tag{2.5.3}$$

which is symmetric as well. To be such the following conditions have to be met [19]:

1. basic function φ has to be *strictly conditionally* positive definite function of order P [18];

2. matrix $\boldsymbol{P}$ has to be full-rank.

The first condition allows a greater freedom on the choice of the basic function, compared to the pure RBF interpolant: in fact, the set of strictly conditionally positive definite functions of order P is a superset of strictly positive definite functions. These functions are defined as those that require a polynomial augmentation of order at least $P-1$ in order to give a non singular $\boldsymbol{M}$. Strictly conditionally positive definite functions of order P are also strictly conditionally positive definite of any higher order. This means that, in case u^h include a polynomial augmentation of order 1, we can also use Multiquadratic, Thin plate splines with $l = 0$ and Polyharmonics with $l = 1$ (refer again to table 2.1 for their definitions) as basic functions. At this point someone might consider increasing the degree P of the polynomial augmentation up to the theoretical limit for the size of matrix $\boldsymbol{P}$, i.e. $M = N$, in order to increase the accuracy of the interpolant. However doing so results in ill-conditioning and singularity issues related to $\boldsymbol{P}$ which in turn will have an impact on the singularity of matrix $\boldsymbol{M}$.

Therefore, since we require the non singularity of $\boldsymbol{M}$, the second condition has to be satisfied. It can be shown that to have a full-rank $\boldsymbol{P}$ the set $\mathcal{X}$, containing the nodes respect to which we are carrying out the interpolation, must be P-unisolvent [18], where P is the degree of the polynomial augmentation. This dependency of $\boldsymbol{P}$'s rank on the node locations should not surprise since the matrix columns consist on the elements of the polynomial basis evaluated at the different points in $\mathcal{X}$, and they are required to be linearly independent.

Given the node generation technique used in this work, a safe rule for a stable implementation (in the sense of a well-posed interpolation problem) of the polynomial augmentation is to respect the inequality $2M \leq N$ where M is the number of terms in the polynomial basis and N is the number of nodes in $\mathcal{X}$.

2.6 Radial Basis Function generated Finite Differences (RBF-FD)

In this section we are going to discuss in detail how Radial Basis Function generated Finite Differences (RBF-FD) meshless method is used to solve Partial Differential Equations (PDEs). Its first implementation was developed by Kansa in [6, 7]. His approach, that we denote as *global method*, due to its computational inefficiencies (that we are going to reference later on this section), was eventually dropped in preference for the one suggested by Tolstykh in [20]. We will denote the latter as *local method*. In recent years RBF-FD local method has been developed and applied with success [21, 22, 23, 24].

The goal of both methods is to approximate solutions to PDEs, i.e., to find a function (or some discrete approximation to this function) which satisfies a given relationship between various of its derivatives on some given region of space along with some boundary conditions on the boundary of this domain. In most cases an analytical solution cannot be found. What RBF-FD methods do is replacing derivatives in the differential equation by Finite Difference (FD) approximations in such a way as to obtain a large algebraic system of equations to be solved in place of the differential equation; this could be easily solved with a computer.

2.6.1 Finite difference method

Before tackling the approximation of PDEs solution, we first consider the more basic task of approximating the derivatives of a known function by Finite Difference (FD) formulas based only on values of the function itself at discrete points. Given u, in the simplest case a function of one variable assumed to be sufficiently smooth, we want to approximate its derivatives at a given point $\overline{x}$ relying solely on its values at a finite number of points close to $\overline{x}$. In general its k-th derivative is approximated by the following FD formula:

$$\left.\frac{d^k u}{dx^k}\right|_{x=\overline{x}} = u^{(k)}(\overline{x}) \approx \sum_{i=1}^{n} c_i^k u(x_i) \tag{2.6.1}$$

where $u(x_1), \ldots, u(x_n)$ are the function's samples and $c_1^k, \ldots, c_n^k$, which can be computed in different ways such as the method of undetermined coefficients or via polynomial interpolation [25], are called FD weights.

To give a concrete example we could approximate $u'(\overline{x})$ with the following one-sided approximations:

$$D_+ u(\overline{x}) = \frac{u(\overline{x}+h) - u(\overline{x})}{h} \tag{2.6.2a}$$

$$D_- u(\overline{x}) = \frac{u(\overline{x}) - u(\overline{x}-h)}{h} \tag{2.6.2b}$$

for some value of h. This is motivated by the standard definition of the derivative as the limiting value of this expression as $h \to 0$. In these cases the same FD weights, $\{\, 1/h, -1/h \,\}$, are associated to function values coming from different samples: $\{\, u(\overline{x}+h), u(\overline{x}) \,\}$ for equation (2.6.2a) and $\{\, u(\overline{x}), u(\overline{x}-h) \,\}$ for (2.6.2b). Another possibility is to use the centered approximation:

$$D_0 u(\overline{x}) = \frac{u(\overline{x}+h) - u(\overline{x}-h)}{h} = \frac{1}{2}\big(D_+ u(\overline{x}) + D_- u(\overline{x})\big) \tag{2.6.3}$$

To derive approximations to higher order derivatives, besides the two method mentioned above, is also possible to repeatedly apply first order differences. Just

as the second order derivatives is the derivative of u', we can view $D^2u(\overline{x})$, the second order derivative approximant, as being a finite difference of first differences: $D^2u(\overline{x}) = D_+D_-u(\overline{x})$ or $D^2u(\overline{x}) = D_-D_+u(\overline{x})$. If we use a step size $h/2$ in each centered approximation to the first derivative we could also define $D^2u(\overline{x})$ as a centered difference of centered differences and obtain:

$$D^2u(\overline{x}) = \frac{1}{h}\left(\left(\frac{u(\overline{x}+h)-u(\overline{x})}{h}\right) - \left(\frac{u(\overline{x})-u(\overline{x}-h)}{h}\right)\right) \tag{2.6.4}$$

where FD weights $\{\, 1/h^2, -2/h^2, 1/h^2 \,\}$ are associated to $\{\, u(\overline{x}+h), u(\overline{x}), u(\overline{x}-h) \,\}$.

2.6.2 Formulation

After this very brief introduction to finite difference (FD), we can see how they are generalized to PDEs differential operators by RBF-FD in order to obtain easily solvable linear systems.

We start by recalling that the boundary value problem that we have to solve is defined as:

$$\begin{cases} \mathcal{L}u = f & \text{in } \Omega \\ \mathcal{B}u = g & \text{on } \partial\Omega \end{cases} \tag{2.6.5}$$

where $\mathcal{L}$ and $\mathcal{B}$ are linear operators and its solution u could be approximated by a function u^h defined as reported in equation (2.4.1):

$$u^h(\boldsymbol{x}) = \sum_{j=1}^{N} \alpha_j \Phi_j(\boldsymbol{x}) + \sum_{k=1}^{M} \beta_k p_k(\boldsymbol{x}) \tag{2.6.6}$$

which holds true all over the domain. Coefficients of the expansions are found by solving equation (2.4.3):

$$\underbrace{\begin{bmatrix} \boldsymbol{B} & \boldsymbol{P} \\ \boldsymbol{P}^T & \boldsymbol{0} \end{bmatrix}}_{\boldsymbol{M}} \begin{bmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{0} \end{bmatrix} \tag{2.6.7}$$

In order to streamline the explanation of the procedure we split the set of nodes $\mathcal{X}$ reported in (2.0.1) into the two following sets:

$$\mathcal{X}_I := \{\, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{N_I} \mid \boldsymbol{x}_i \in \Omega,\ i = 1, \ldots, N_I \,\} \tag{2.6.8a}$$

$$\mathcal{X}_B := \{\, \boldsymbol{x}_{N_{I+1}}, \ldots, \boldsymbol{x}_{N_B} \mid \boldsymbol{x}_i \in \partial\Omega,\ i = N_{I+1}, \ldots, N_B \,\} \tag{2.6.8b}$$

where N_I and N_B indicate respectively the number of nodes inside and on the boundary of the physical domain (from the definitions of the two sets it also follows that $N_I + N_B = N$ and $\mathcal{X}_I \cup \mathcal{X}_B = \mathcal{X}$).

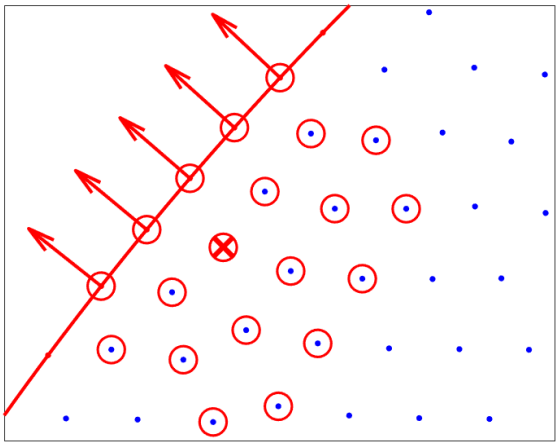


Figure 2.1: Example of 2D stencil. The nodes which belong to it are marked with a red circle, its central node with a red cross. Red arrows represent the boundary normals $\boldsymbol{n}$ of those nodes belonging to $\partial\Omega$. Figure taken from [19]

The idea that Kansa used to discretize the problem reported above is to use the theory of interpolation to approximate the differential operator $\mathcal{L}$ in the PDE, with an operator $\mathcal{L}^h$ represented by a matrix $\boldsymbol{C}_I$, using a FD-like method. Tolstykh's approach follows the same concept as Kansa's, but adds *stencil* as novelty.

A stencil of m nodes associated to each node $\boldsymbol{x}_i \in \mathcal{X}_I$ is defined as the set $\mathcal{X}_i = \{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_m \} \subseteq \mathcal{X}$ formed by the m nearest neighbors of $\boldsymbol{x}_i$; in addition $\mathcal{X}_i$ can be interpreted as the union of $\mathcal{X}_{i,I} = \{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_{m_I} \}$ and $\mathcal{X}_{i,B} = \{ \boldsymbol{x}_{m_I+1}, \dots, \boldsymbol{x}_m \}$, respectively the set of its m_I nodes belonging to Ω and of its other m_B nodes belonging to $\partial\Omega$. An example of stencil could be found in Figure 2.1.

In Tolstykh's local method the interpolation scheme is local, i.e. u^h is expanded using a basis that changes depending on the position $\boldsymbol{x}$ and it is made valid only inside the stencil centered at $\boldsymbol{x}$ rather than globally across the entire domain. Thus, given a point $\boldsymbol{x}_i \in \mathcal{X}_I$ along its related stencil $\mathcal{X}_i = \{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_m \}$, equation (2.6.6) is rewritten as:

$$u^h(\boldsymbol{x}_i) = \sum_{j=1}^{m} \alpha_j \Phi(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{k=1}^{M} \beta_k p_k(\boldsymbol{x}_i) \tag{2.6.9}$$

In the following, we will focus solely on Tolstykh's formulation, commonly employed in practice, with a brief mention of Kansa's for comparative purposes.

Applying operator $\mathcal{L}$ to the definition of u^h results in:

$$\begin{aligned}\mathcal{L}u^h(\boldsymbol{x}_i) &= \sum_{j=1}^{m} \alpha_j \mathcal{L}\Phi(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{k=1}^{M} \beta_k \mathcal{L}p_k(\boldsymbol{x}_i) \\ &= \begin{bmatrix}\boldsymbol{\alpha} & \boldsymbol{\beta}\end{bmatrix} \begin{bmatrix}\mathcal{L}\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i)\end{bmatrix}\end{aligned} \tag{2.6.10}$$

where different vectors of coefficients $\boldsymbol{\alpha} = \begin{bmatrix}\alpha_1, \dots, \alpha_m\end{bmatrix} \in \mathbb{R}^m$ and $\boldsymbol{\beta} = \begin{bmatrix}\beta_1, \dots, \beta_M\end{bmatrix} \in \mathbb{R}^M$ are associated to different $\boldsymbol{x}_i$. In order to find them we can note that, since equation (2.6.9) is still an approximated solution of the boundary value problem (at least locally), it must satisfy:

$$u^h(\boldsymbol{x}_i) = u(\boldsymbol{x}_i) \quad \text{if } \boldsymbol{x}_i \in \mathcal{X}_{i,I} \tag{2.6.11a}$$

$$\mathcal{B}u^h(\boldsymbol{x}_i) = g(\boldsymbol{x}_i) \quad \text{if } \boldsymbol{x}_i \in \mathcal{X}_{i,B} \tag{2.6.11b}$$

which rewritten in matrix form read as:

$$\underbrace{\begin{bmatrix}\boldsymbol{\Phi}_I & \boldsymbol{P}_I \\ \boldsymbol{\mathcal{B}\Phi}_B & \boldsymbol{\mathcal{B}P}_B \\ \boldsymbol{P}^T & \boldsymbol{0}\end{bmatrix}}_{\boldsymbol{M}_{BC}} \begin{bmatrix}\boldsymbol{\alpha} \\ \boldsymbol{\beta}\end{bmatrix} = \begin{bmatrix}\boldsymbol{u}_I \\ \boldsymbol{g} \\ \boldsymbol{0}\end{bmatrix} \tag{2.6.12}$$

where $\boldsymbol{u}_I = [u(\boldsymbol{x}_1), \dots, u(\boldsymbol{x}_{m_I})]^T \in R^{m_I}$ and $\boldsymbol{g} = [g(\boldsymbol{x}_{m_I+1}), \dots, g(\boldsymbol{x}_m)]^T \in \mathbb{R}^{m_B}$ and the new terms in $\boldsymbol{M}_{BC} \in \mathbb{R}^{(m+M)\times(m+M)}$ are defined as follow:

$$\begin{aligned}
\boldsymbol{\Phi}_I &= \begin{bmatrix}\Phi(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & \Phi(\boldsymbol{x}_1, \boldsymbol{x}_m) \\ \vdots & \ddots & \vdots \\ \Phi(\boldsymbol{x}_{m_i}, \boldsymbol{x}_1) & \dots & \Phi(\boldsymbol{x}_{m_I}, \boldsymbol{x}_m)\end{bmatrix} \in \mathbb{R}^{m_I \times m} \\
\boldsymbol{P}_I &= \begin{bmatrix}p_1(\boldsymbol{x}_1) & \dots & p_M(\boldsymbol{x}_{m_I}) \\ \vdots & \ddots & \vdots \\ p_1(\boldsymbol{x}_1) & \dots & p_M(\boldsymbol{x}_{m_I})\end{bmatrix} \in \mathbb{R}^{m_I \times M} \\
\boldsymbol{\mathcal{B}\Phi}_B &= \begin{bmatrix}\mathcal{B}\Phi(\boldsymbol{x}_{m_I+1}, \boldsymbol{x_1}) & \dots & \mathcal{B}\Phi(\boldsymbol{x}_{m_I+1}, \boldsymbol{x}_m) \\ \vdots & \ddots & \vdots \\ \mathcal{B}\Phi(\boldsymbol{x}_m, \boldsymbol{x_1}) & \dots & \mathcal{B}\Phi(\boldsymbol{x}_m, \boldsymbol{x}_m)\end{bmatrix} \in \mathbb{R}^{m_B \times m} \\
\boldsymbol{\mathcal{B}P}_B &= \begin{bmatrix}\mathcal{B}p_1(\boldsymbol{x}_{m_I+1}) & \dots & \mathcal{B}p_M(\boldsymbol{x}_{m_I+1}) \\ \vdots & \ddots & \vdots \\ \mathcal{B}p_1(\boldsymbol{x}_m) & \dots & \mathcal{B}p_M(\boldsymbol{x}_m)\end{bmatrix} \in \mathbb{R}^{m_B \times M}
\end{aligned} \tag{2.6.13}$$

When Dirichlet BC are enforced, $\mathcal{B}$ becomes the identity operator in which case $\boldsymbol{M}_{BC}$ in equation (2.6.12) coincides with the matrix $\boldsymbol{M}$ of the interpolation system (2.4.3) defined in section 2.4. In this case the problem is well-posed, regardless of the shape of domain Ω if the instructions presented in section 2.5 are adhered to. However this is no-longer true in case of Neumann or Robin BCs: we are going to tackle this issue in the next section. For the time being, to proceed with the explanation of RBF-FD, we will assume that the problem is always well-posed regardless of the type of BCs applied.

Substituting the recently determined $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ into equation (2.6.10) we obtain:

$$\mathcal{L}u^h(\boldsymbol{x}_i) = \begin{bmatrix}\boldsymbol{u}_I & \boldsymbol{g} & \boldsymbol{0}\end{bmatrix} \boldsymbol{M}_{BC}^{-T} \begin{bmatrix}\mathcal{L}\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i)\end{bmatrix} \tag{2.6.14}$$

We might now observe that the last two factors of this last equation are known, thus we can represent their product with a vector $\boldsymbol{c}(\boldsymbol{x}_i) = \begin{bmatrix}\boldsymbol{c}_I(\boldsymbol{x}_i), \boldsymbol{c}_B(\boldsymbol{x}_i), \boldsymbol{c}_p(\boldsymbol{x}_i)\end{bmatrix} \in \mathbb{R}^{m+M}$ that can be obtained solving the dual problem:

$$\boldsymbol{M}_{BC}^T \begin{bmatrix}\boldsymbol{c}_I(\boldsymbol{x}_i) \\ \boldsymbol{c}_B(\boldsymbol{x}_i) \\ \boldsymbol{c}_p(\boldsymbol{x}_i)\end{bmatrix} = \begin{bmatrix}\mathcal{L}\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i)\end{bmatrix} \tag{2.6.15}$$

where $\boldsymbol{c}_I(\boldsymbol{x}_i)$, $\boldsymbol{c}_B(\boldsymbol{x}_i)$ and $\boldsymbol{c}_p(\boldsymbol{x}_i)$ denote the first m_I, the next m_B and the last M elements of $\boldsymbol{c}(\boldsymbol{x}_i)$ respectively. Once this is done (2.6.14) can be rewritten as:

$$\begin{aligned}\mathcal{L}u^h(\boldsymbol{x}_i) &= \boldsymbol{c}_I(\boldsymbol{x}_i)^T \boldsymbol{u}_I + \boldsymbol{c}_B(\boldsymbol{x}_i)^T \boldsymbol{g} \\ &= \sum_{j=1}^{m_I} c_j(\boldsymbol{x}_i) u(\boldsymbol{x}_j) + \sum_{k=m_I+1}^{m} c_k(\boldsymbol{x}_i) g(\boldsymbol{x}_k)\end{aligned} \tag{2.6.16}$$

which is nothing else but the FD-like approximation of $\mathcal{L}u^h$ at point $\boldsymbol{x}_i$ with FD weights given by the elements of the vectors $\boldsymbol{c}_I(\boldsymbol{x}_i)$ and $\boldsymbol{c}_P(\boldsymbol{x}_i)$. In general the approximation is not exact, however it is in this specific case thanks to the combination of:

- the linearity of u^h respect the parameters $\alpha_1, \ldots, \alpha_m$ and $\beta_1, \ldots, \beta_M$, as can be seen in equation (2.6.9), and;
- the linearity of operator $\mathcal{L}$.

The values of the approximated solution at the N_I nodes of $\mathcal{X}$, i.e. the solution of the PDE, can be finally found by requiring u^h to approximate the exact solution at each of these points:

$$\mathcal{L}u^h(\boldsymbol{x}_i) = \mathcal{L}u(\boldsymbol{x}_i) = f(\boldsymbol{x}_i) \quad \text{if } \boldsymbol{x}_i \in \mathcal{X}_{i,I} \tag{2.6.17}$$

which takes the matrix form:

$$\boldsymbol{C}_I \begin{bmatrix} u^h(\boldsymbol{x}_1) \\ \vdots \\ u^h(\boldsymbol{x}_{N_I}) \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{x}_1) \\ \vdots \\ f(\boldsymbol{x}_{N_I})) \end{bmatrix} - \boldsymbol{C}_B \begin{bmatrix} g(\boldsymbol{x}_{N_I+1}) \\ \vdots \\ g(\boldsymbol{x}_N) \end{bmatrix} \tag{2.6.18}$$

where rows of matrices $\boldsymbol{C}_I$ and $\boldsymbol{C}_B$ are formed by the elements of the vectors $\boldsymbol{c}_I(\boldsymbol{x}_i)$ and $\boldsymbol{c}_B(\boldsymbol{x}_i)$ found by solving solving equation (2.6.15) N_I times:

$$\begin{aligned} \boldsymbol{C}_I &= \begin{bmatrix} c_1(\boldsymbol{x}_1) & \dots & c_{N_I}(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ c_1(\boldsymbol{x}_{N_I}) & \dots & c_{N_I}(\boldsymbol{x}_{N_I}) \end{bmatrix} \in \mathbb{R}^{N_I \times N_I} \\ \boldsymbol{C}_B &= \begin{bmatrix} c_{N_I+1}(\boldsymbol{x}_1) & \dots & c_N(\boldsymbol{x}_1) \\ \vdots & \ddots & \vdots \\ c_{N_I+1}(\boldsymbol{x}_{N_I}) & \dots & c_N(\boldsymbol{x}_{N_I}) \end{bmatrix} \in \mathbb{R}^{N_I \times N_B} \end{aligned} \tag{2.6.19}$$

What we just obtained in (2.6.18) is the algebraic system that once solved, in place of the PDE, gives the values of the PDE approximated solution $\{ u^h(\boldsymbol{x}_1), \dots, u^h(\boldsymbol{x}_{N_I}) \}$. In Kansa's global method the process is followed with the same steps as long as the global expansion of equation (2.6.6) is used to approximate the PDE solution instead of the local expansion of equation (2.6.9).

An important remark has to be done about computational efficiencies of global and local methods since linear system (2.6.15) has to be solved at any point $\boldsymbol{x}_i \in \mathcal{X}_I$, therefore requiring the inversion of N_I different $\boldsymbol{M}_{BC}^T$ matrices. In case of local method each $\boldsymbol{M}_{BC}$ has size $(m+M) \times (m+M)$; instead in global method they have size $(N+M) \times (N+M)$ since conditions (2.6.11) must be enforced not only in the stencil, but over the entire domain due to the global nature of the approximated solution. Remembering that the computational cost of factorizing a matrix is at least of $\mathcal{O}(n^3)$, where here n denotes the number of rows and columns of the matrix, then local method is computationally advantageous respect the global one.

Another difference between global and local method is that in global method derivative approximation of u^h at point $\boldsymbol{x}_i$ depends on the values of u across the entire domain even if derivatives are local properties of functions: this is clearly suboptimal since leads to a matrix $\boldsymbol{C}_I$, which represent the discretized differential operator $\mathcal{L}$, that is full even though it should not be in principle. On the contrary, in local method, $\boldsymbol{C}_I$ is sparse since in each row i the number of non-zero entries is equal to the number of internal nodes of stencil $\mathcal{X}_I$.

Finally, indicating with $\boldsymbol{u}^h = [u(\boldsymbol{x}_1), \dots, u(\boldsymbol{x}_N)]$ the vector of the exact solution evaluated at the RBF-FD nodes and with $\boldsymbol{u} = \left[u^h(\boldsymbol{x}_1), \dots, u^h(\boldsymbol{x}_N)\right]$ the

approximated solution obtained from RBF-FD method, is possible to show that the solution error $e(N) = \frac{\|\boldsymbol{u}^h - \boldsymbol{u}\|}{\|\boldsymbol{u}\|}$ decrease exponentially as the number of nodes N increase in the global approach. In the local approach, conversely, the solution error decreases "only" polynomially.

The reason just listed above are those that explain the benefits of stencil introduction and that led to the choice of Tolstykh approach both in this work and within the community of researchers exploring different Meshless Methods.

What we have not covered yet is the computational cost of the stencils creation: if too expensive, it might loose all the benefits of the local approach. In fact this would be the case when using a brute force approach (which require a computational cost of $\mathcal{O}(N^2)$ for each $\boldsymbol{x}_i$) where all pairwise distances between nodes are computed and then sorted in order to keep just the m-nearest neighbors. To avoid this issue more efficient algorithms has to be used. An example of these is the k-d tree algorithm [26] which require $\mathcal{O}(N \log N)$ operations for rearranging the nodes and other $\mathcal{O}(N \log N)$ for finding a fixed number of neighbors.

Finally, regarding to the stencil, we note that the effect of its size are still under research: one of the last activities on this regard is the one of Kolar-Požun et al. [27] where they found that changing stencil's size induce oscillations in both the solution and discretization errors for the Poisson equation.

2.7 Radial Basis Functions-Hermite Finite Difference (RBF-HFD)

In previous section we have overlooked a rather important fact for the implementation of the RBF-FD method: the invertibility of matrix $\boldsymbol{M}_{BC}$. In fact is always possible, given a stencil containing boundary points on which Neumann or Robin Boundary Conditions (BCs) are applied, to find normal directions associated with them that make the local natrix $\boldsymbol{M}_{BC}$ in equation (2.6.12) singular [19].

Unfortunately, simply avoiding this kind of boundary condition is not an option due to their importance in most boundary value problems of engineering interest. The solution adopted in the present work to avoid this limitation is the so called Radial Basis Functions-Hermite Finite Difference (RBF-HFD). Within this section we present the formulation of the aforementioned method after explaining the Hermite interpolation framework which is at its core.

2.7.1 Hermite Interpolation

In the RBF-FD method, to find the approximated solution u^h valid for a given stencil $\mathcal{X}_q = \{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_m \} \subseteq \mathcal{X}$, based on a point $\boldsymbol{x}_q \in \Omega$, the following

interpolation conditions are enforced:

$$u^h(\boldsymbol{x}_i) = u(\boldsymbol{x}_i) \quad \text{if } \boldsymbol{x}_i \in \mathcal{X}_{q,I} \tag{2.7.1a}$$

$$\mathcal{B}u^h(\boldsymbol{x}_i) = g(\boldsymbol{x}_i) \quad \text{if } \boldsymbol{x}_i \in \mathcal{X}_{q,B} \tag{2.7.1b}$$

In general the action of evaluating a function u at a point $\boldsymbol{x}_i$ could be seen as the application of an operator $\delta_{\boldsymbol{x}_i}$, called point-evaluation functional, such that $\delta_{\boldsymbol{x}_i} u = u(\boldsymbol{x}_i)$. Furthermore functional $\delta_{\boldsymbol{x}_i}$ can be combined with other operators; an example of this can be found when Neumann BC are encountered in conditions (2.7.1b):

$$\begin{aligned} \mathcal{B}u^h(\boldsymbol{x}_i) &= \frac{\partial}{\partial \boldsymbol{n}} u^h(\boldsymbol{x}_i) \\ &= \left(\delta_{\boldsymbol{x}_i} \circ \frac{\partial}{\partial \boldsymbol{n}} \right) u^h(\cdot) \end{aligned} \tag{2.7.2}$$

where $\boldsymbol{x}_i$ belongs to $\mathcal{X}_{q,B_N} \subseteq \mathcal{X}_{q,B}$, the set of stencil's boundary nodes on which are applied Neumann BCs. To avoid excessive notation complexity in the following we assume that only Neumann BC are enforced on the points of the stencil that belongs to the boundary of the physical domain, i.e. $\mathcal{X}_{q,B_N} \equiv \mathcal{X}_{q,B}$.

Leveraging on these operators, interpolation conditions, more generally, can be defined through a set of given values $\{ u(\boldsymbol{x}_1), \dots, u(\boldsymbol{x}_m) \}$ along with a given set of functionals $\Lambda = \{ \lambda_1, \dots, \lambda_m \}$ as follow:

$$\lambda_i u^h = \lambda_i u \qquad i = 1, \dots, m \tag{2.7.3}$$

These conditions are typically called Hermite interpolation conditions [18]. For RBF-FD local constraints (2.7.1) we have that $\lambda_i = \delta_{\boldsymbol{x}_i}$ for those nodes $\boldsymbol{x}_i$ that belong to $\mathcal{X}_{q,I}$ and $\lambda_i = \delta_{\boldsymbol{x}_i} \circ \frac{\partial}{\partial \boldsymbol{n}}$ for those which belong to $\mathcal{X}_{q,B}$.

In case of pure RBF interpolation scheme the local approximated solution u^h is given by:

$$u^h(\cdot) = \sum_{j=1}^{m} \alpha_j \Phi(\cdot, \boldsymbol{x}_j) \tag{2.7.4}$$

where "$\cdot$" is a placeholder for the argument of the function meaning that it is not evaluated at any point. In this case the basis used for the interpolation is determined solely by the points of the stencil and is given by $\{ \Phi(\cdot, \boldsymbol{x}_1), \dots, \Phi(\cdot, \boldsymbol{x}_m) \} = \{ \delta_{2,\boldsymbol{x}_1} \Phi(\cdot,\cdot), \dots, \delta_{2,\boldsymbol{x}_m} \Phi(\cdot,\cdot) \}$ where the subscript 2 in the point-evaluation functional indicates that it is acting on the second argument of $\Phi(\cdot,\cdot)$. Using interpolant (2.7.4) in case of Hermite interpolation conditions and repeating the same scattered data interpolation procedure described in section 2.1, we would obtain a matrix $\boldsymbol{B}$:

$$\boldsymbol{B} = \begin{bmatrix} \lambda_1 \Phi(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & \lambda_1 \Phi(\boldsymbol{x}_1, \boldsymbol{x}_m) \\ \vdots & \ddots & \vdots \\ \lambda_m \Phi(\boldsymbol{x}_m, \boldsymbol{x}_1) & \dots & \lambda_m \Phi(\boldsymbol{x}_m, \boldsymbol{x}_m) \end{bmatrix} \tag{2.7.5}$$

which is not symmetrical and does not satisfy, in general, the conditions of positive definiteness or non-singularity. This is exactly the case when RBF interpolation is applied to stencil $\mathcal{X}_q$ where $\boldsymbol{B}$ becomes:

$$\boldsymbol{B} = \begin{bmatrix} \delta_{1,\boldsymbol{x}_1}\Phi(\cdot,\boldsymbol{x}_1) & \dots & \delta_{1,\boldsymbol{x}_1}\Phi(\cdot,\boldsymbol{x}_m) \\ \vdots & \ddots & \vdots \\ \delta_{1,\boldsymbol{x}_{m_I}}\Phi(\cdot,\boldsymbol{x}_1) & \dots & \delta_{1,\boldsymbol{x}_{m_I}}\Phi(\cdot,\boldsymbol{x}_m) \\ \delta_{1,\boldsymbol{x}_{m_I+1}} \circ \frac{\partial}{\partial \boldsymbol{n}}\Phi(\cdot,\boldsymbol{x}_1) & \dots & \delta_{1,\boldsymbol{x}_{m_I+1}} \circ \frac{\partial}{\partial \boldsymbol{n}}\Phi(\cdot,\boldsymbol{x}_m) \\ \vdots & \ddots & \vdots \\ \delta_{1,\boldsymbol{x}_m} \circ \frac{\partial}{\partial \boldsymbol{n}}\Phi(\cdot,\boldsymbol{x}_1) & \dots & \delta_{1,\boldsymbol{x}_m} \circ \frac{\partial}{\partial \boldsymbol{n}}\Phi(\cdot,\boldsymbol{x}_m) \end{bmatrix} \tag{2.7.6}$$

which is not symmetric due to the rows associated to $\delta_{1,\boldsymbol{x}_i} \circ \frac{\partial}{\partial \boldsymbol{n}}$ operators obtained from Neumann BCs. We remark once more that $\boldsymbol{B}$ would have remained symmetrical in the case of Dirichlet BCs only, since their associated operator would have been just $\delta_{1,\boldsymbol{x}_i}$, the same applied on the first m_I rows.

An improvement, is done to the basic RBF interpolant in (2.7.4) in order to solve this problem: a more general interpolation basis is constructed by applying different functionals other than simple point evaluation to the same kernel $\Phi(\cdot,\cdot)$ (where we recall that a kernel is simply a real valued function defined on a space given by the Cartesian product $\Omega \times \Omega$); this leads to the so-called *Generalized Hermite Interpolant*. Suppose that the operator $\delta_{2,\boldsymbol{x}_j}$ of the canonical RBF base is replaced with $\lambda_{2,j}$, the functional acting on the j-th Hermite condition, then the interpolation basis, in general, becomes $\{\, \lambda_{2,1}\Phi(\cdot,\cdot), \dots, \lambda_{2,m}\Phi(\cdot,\cdot) \,\}$. In contrast to the classical scheme, the basis now does not depend solely on the nodes of the stencil to which Φ is applied, but also on the conditions associated with them: this is the key feature that differentiate this approach compared to the simple RBF-FD one.

The new interpolant u^h defined according to the generalized Hermite interpolation scheme then becomes:

$$u^h(\cdot) = \sum_{j=1}^{m} \alpha_j \lambda_{2,j}\Phi(\cdot,\cdot) \tag{2.7.7}$$

which, once it is applied for interpolation, leads to an interpolation matrix:

$$\boldsymbol{B}_H = \begin{bmatrix} \lambda_{1,1}\lambda_{2,1}\Phi(\cdot,\cdot) & \dots & \lambda_{1,1}\lambda_{2,m}\Phi(\cdot,\cdot) \\ \vdots & \ddots & \vdots \\ \lambda_{1,m}\lambda_{2,1}\Phi(\cdot,\cdot) & \dots & \lambda_{1,m}\lambda_{2,m}\Phi(\cdot,\cdot) \end{bmatrix} \tag{2.7.8}$$

which is symmetric if $\lambda_{1,j} = \lambda_{2,j}$ for $j = 1, \dots, m$. Because of this property, Hermite's formulation is also called *symmetric*, in contrast to the classical one

which is referred to *unsymmetryc*. Furthermore $\boldsymbol{B}_H$ is also positive definite when the basic function φ used to define the kernel is strictly positive definite and operators λ_j are linearly independents [19]. The symmetry and positive definiteness properties previously mentioned also hold more generally when the Hermite interpolation conditions in equation (2.7.1) involves not only Neumann BCs, but also Dirichlet or Robin BCs, i.e. when $\mathcal{X}_{q,B_N} \not\equiv \mathcal{X}_{q,B}$.

In addition, adoption of polynomial augmentation, as explained in section 2.4, to increase the accuracy of the RBF interpolant is still possible on condition of some changes. The augmented Hermite interpolant become:

$$u^h(\cdot) = \sum_{j=1}^{m} \alpha_j \lambda_{2,j} \Phi(\cdot,\cdot) + \sum_{k=1}^{M} \beta_k p_k(\cdot) \tag{2.7.9}$$

In order to obtain a solvable linear system to find coefficients $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, conditions (2.4.2) are modified as follow:

$$\sum_{i=1}^{m} \alpha_i \lambda_i p_k(\cdot) = 0, \qquad i = 1, \dots, M \tag{2.7.10}$$

thus leading to:

$$\underbrace{\begin{bmatrix} \boldsymbol{B_H} & \boldsymbol{P}_H \\ \boldsymbol{P}_H^T & \boldsymbol{0} \end{bmatrix}}_{\boldsymbol{M}} \begin{bmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Lambda u} \\ \boldsymbol{0} \end{bmatrix} \tag{2.7.11}$$

where $\boldsymbol{\Lambda u} = [\lambda_1 u, \dots, \lambda_m u]$ and $\boldsymbol{P}_H$ is defined as:

$$\boldsymbol{P}_H = \begin{bmatrix} \lambda_1 p_1(\cdot) & \dots & \lambda_1 p_M(\cdot) \\ \vdots & \ddots & \vdots \\ \lambda_m p_1(\cdot) & \dots & \lambda_m p_M(\cdot) \end{bmatrix} \tag{2.7.12}$$

2.7.2 Formulation

The Hermite interpolant can be adopted also in Tolstykh's local approach with some modifications, that we explain in this subsection. We remember once again that the problem that has to be solved is:

$$\begin{cases} \mathcal{L}u = f & \text{in } \Omega \\ \mathcal{B}u = g & \text{on } \partial\Omega \end{cases} \tag{2.7.13}$$

and for each internal node $\boldsymbol{x}_i$ a stencil $\mathcal{X}_i = \{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_m \}$ composed of its m nearest neighbors is constructed. $\mathcal{X}_i$ is then splitted in $\mathcal{X}_{i,I} = \{ \boldsymbol{x}_1, \dots, \boldsymbol{x}_{m_I} \}$ and

$\mathcal{X}_{i,B} = \{ \boldsymbol{x}_{m_I+1}, \dots, \boldsymbol{x}_m \}$ respectively the set of stencil points that belong to the interior and boundary of the physical domain Ω.

Interpolation conditions inside the stencil are the same as those reported at the beginning of previous subsection:

$$u^h(\boldsymbol{x}_i) = u(\boldsymbol{x}_i) \quad \text{if } \boldsymbol{x}_i \in \mathcal{X}_{q,I} \tag{2.7.14a}$$

$$\mathcal{B}u^h(\boldsymbol{x}_i) = g(\boldsymbol{x}_i) \quad \text{if } \boldsymbol{x}_i \in \mathcal{X}_{q,B} \tag{2.7.14b}$$

At this point the interpolant or local approximated solution of the problem is defined according to the Hermite interpolation theory and takes the same form reported in equation (2.7.9). Making explicit the functionals $\lambda_1, \dots, \lambda_m$ used in the definition of u^h and applying it at the center of the stencil $\mathcal{X}_I$, we can write it as:

$$u^h(\boldsymbol{x}_i) = \sum_{j=1}^{m_I} \alpha_j \Phi(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{j=m_I+1}^{m} \alpha_j \mathcal{B}_2 \Phi(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{k=1}^{M} \beta_k p_k(\boldsymbol{x}_i) \tag{2.7.15}$$

where $\boldsymbol{\alpha}_I = [\alpha_1, \dots, \alpha_{m_I}]$ and $\boldsymbol{\alpha}_B = [\alpha_{m_I+1}, \dots, \alpha_m]$ are used to distinguish the coefficients that are associated to internal and boundary nodes. Conditions (2.7.10) have to be enforced on the coefficients of the expansion in order to obtain a square coefficient matrix associated to constraints (2.7.14):

$$\sum_{j=1}^{m_I} \alpha_j p_k(\boldsymbol{x}_j) + \sum_{j=m_I+1}^{m} \alpha_j \mathcal{B} p_k(\boldsymbol{x}_j) = 0 \qquad k = 1, \dots, M \tag{2.7.16}$$

Due to the new form of the interpolant u^h, system (2.6.12) now becomes:

$$\underbrace{\begin{bmatrix} \boldsymbol{\Phi}_{I,I} & \boldsymbol{\mathcal{B}}_2\boldsymbol{\Phi}_{I,B} & \boldsymbol{P}_I \\ \boldsymbol{\mathcal{B}}_1\boldsymbol{\Phi}_{B,I} & \boldsymbol{\mathcal{B}}_1\boldsymbol{\mathcal{B}}_2\boldsymbol{\Phi}_{B,B} & \boldsymbol{\mathcal{B}}\boldsymbol{P}_B \\ \boldsymbol{P}_I^T & \boldsymbol{\mathcal{B}}\boldsymbol{P}_B^T & \boldsymbol{0} \end{bmatrix}}_{\boldsymbol{M}_{BC}} \begin{bmatrix} \boldsymbol{\alpha}_I \\ \boldsymbol{\alpha}_B \\ \boldsymbol{\beta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{u}_I \\ \boldsymbol{g} \\ \boldsymbol{0} \end{bmatrix} \tag{2.7.17}$$

Proceeding by following the same steps of the RBF-FD method we apply the linear operator $\mathcal{L}$ to the interpolant u^h:

$$\begin{aligned} \mathcal{L}u^h(\boldsymbol{x}_i) &= \sum_{j=1}^{m_I} \alpha_j \mathcal{L}_1 \Phi(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{j=m_I+1}^{m} \alpha_j \mathcal{L}_1 \mathcal{B}_2 \Phi(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sum_{k=1}^{M} \beta_k \mathcal{L} p_k(\boldsymbol{x}_i) \\ &= \begin{bmatrix} \boldsymbol{\alpha}_I & \boldsymbol{\alpha}_B & \boldsymbol{\beta} \end{bmatrix} \begin{bmatrix} \mathcal{L}_1 \boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}_1 \mathcal{B}_2 \boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,B}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i) \end{bmatrix} \end{aligned} \tag{2.7.18}$$

Coefficient vectors $\boldsymbol{\alpha}_I$, $\boldsymbol{\alpha}_B$ and $\boldsymbol{\beta}$ obtained from equation (2.7.17) are then substituted in equation (2.7.18) leading to:

$$\mathcal{L}u^h(\boldsymbol{x}_i) = \begin{bmatrix} \boldsymbol{u}_I & \boldsymbol{g} & \boldsymbol{0} \end{bmatrix} {\boldsymbol{M_{BC}}}^{-T} \begin{bmatrix} \mathcal{L}_1\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}_1\mathcal{B}_2\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,B}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i) \end{bmatrix} \tag{2.7.19}$$

Exactly as in RBF-FD formulation the product of the last two factors of the aforementioned equation is known and the resulting vector $\boldsymbol{c}(\boldsymbol{x}_i) = \begin{bmatrix} \boldsymbol{c}_I(\boldsymbol{x}_i), \boldsymbol{c}_B(\boldsymbol{x}_i), \boldsymbol{c}_p(\boldsymbol{x}_i) \end{bmatrix}$ contains the coefficients that make up the Finite Difference-like approximation of the differential operator $\mathcal{L}$ in a given point $\boldsymbol{x}_i$. In this case $\boldsymbol{c}(\boldsymbol{x}_i)$ is computed by solving:

$${\boldsymbol{M_{BC}}}^{T} \begin{bmatrix} \boldsymbol{c}_I(\boldsymbol{x}_i) \\ \boldsymbol{c}_B(\boldsymbol{x}_i) \\ \boldsymbol{c}_p(\boldsymbol{x}_i) \end{bmatrix} = \begin{bmatrix} \mathcal{L}_1\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}_1\mathcal{B}_2\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,B}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i) \end{bmatrix} \tag{2.7.20}$$

From now on the steps to build the matrix $\boldsymbol{C}_I$, that approximate $\mathcal{L}$, are the same of the local RBF-FD ones. The same applies to obtaining the values of the approximate solution of the partial differential equation at the nodes inside the domain.

Chapter 3

Adjoint method

3.1 Automatic Differentiation (AD)

In general there exists different ways to compute derivatives using a computer program, these are:

Manual Differentiation In this case *analytical* derivatives are computed by hand and then are plugged into standard optimization procedures such as gradient descend. Of course doing so is time consuming and prone to error.

Numerical Differentiation In this case finite difference methods, as the ones reported in subsection 2.6.1, are used to approximate the derivatives *values*. Easy to implement it has the disadvantage to be inaccurate due to round-off and truncation errors [25].

Symbolic Differentiation This case addresses the weakness of both manual and numerical differentiation. *Analytical* derivative expressions are automatically obtained by modern computer algebra systems such as Mathematica[1] or SymPy[2]. Unfortunately, often, the outcomes are plagued with the problem of "expression swell" which means that the resulting expressions are large, complex and cryptic. Furthermore it can be applied only to models defined in closed-form.

Automatic Differentiation This last case refers to a family of techniques that compute derivative *values* (in contrast with symbolic differentiation) by using symbolic rules of differentiation (but keeping track of derivative values as

[1]Wolfram Research, Inc.'s proprietary software for technical computing. See https://www.wolfram.com/mathematica/ for more informations.

[2]Python open source library for symbolic mathematics. The project can be found at https://www.sympy.org/en/index.html.

opposed to the resulting expressions) through accumulation of values during code execution. Thanks to this, automatic differentiation, can be applied to code involving branches, loops and recursion as opposed to symbolic differentiation [28]. The mix between symbolic and numerical differentiation gives to these methodologies an hybrid nature.

For the remaining of this section we will explain the details of Automatic Differentiation (AD), to do so we will look at its two (main) implementations:

- *forward mode*, also known as tangent linear mode; and
- *reverse mode* also known as cotangent linear mode or *adjoint* mode.

To explain how each of the two modes works, we show how they are applied to a function $\tilde{f}\colon \mathbb{R}^2 \to \mathbb{R}^2$ defined as:

$$\tilde{f}(x_1, x_2) = \begin{bmatrix} \tilde{f}_1(x_1, x_2) & \tilde{f}_2(x_1, x_2) \end{bmatrix} \tag{3.1.1}$$

with $\tilde{f}_1(x_1, x_2) = x_1 x_2 + \cos x_1$ and $\tilde{f}_2(x_1, x_2) = x_2^3 + \ln x_1 - x_2$.

Before entering into the detail, we notice that every function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ can be rewritten as a computational graph. A computational graph is a direct graph whose nodes correspond to operations and each operation can feed its output into other operations; once the graph is fed with some variables each node become automatically function of those variables. The usual operations considered as nodes are: binary arithmetic operations, the unary sign switch and transcendental functions such as exponential, logarithm and trigonometric functions. Creating a computational graph for a function becomes easy by indicating with:

- $v_{i-n} = x_i$, $i = 1, \dots, n$ the input variables;
- v_i, $i = 1, \dots, l$ the intermediate variable;
- $y_{m-i} = v_{l-i}$, $i = m-1, \dots, 0$ the output variables.

To better identify the relationship between the elementary operations which constitute a function it is helpful creating an *evaluation trace* for the function itself. Left-hand side of table 3.1 and figure 3.1 contains respectively the evaluation trace and the computational graph for function $\tilde{f}$.

Using the aforementioned representations we see that every function ultimately is a composition of elementary operations. This also means that its numerical derivatives can be computed by combining all the numerical derivatives of the constituent operations through the *chain rule*: this, is the main idea of Automatic Differentiation (AD).

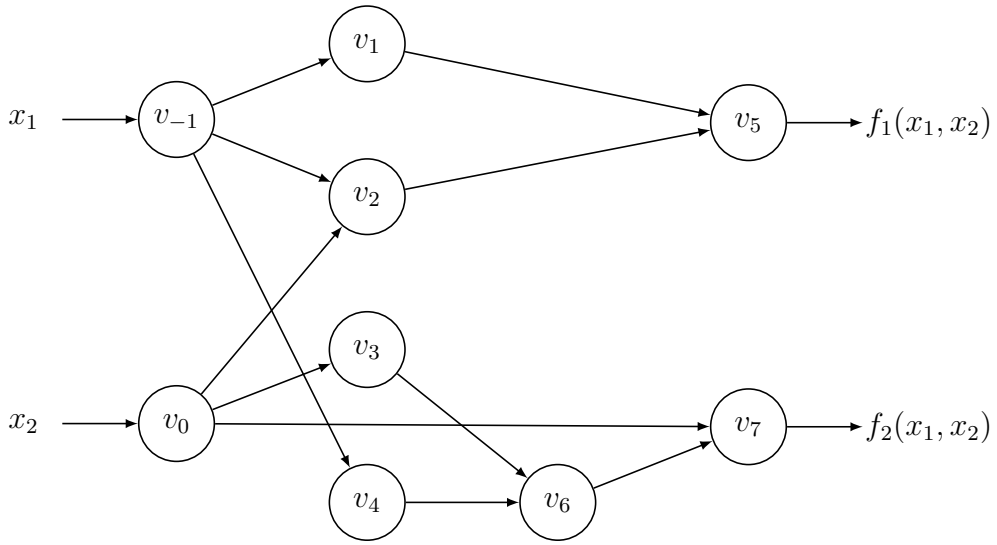


Figure 3.1: Computational graph of function $\tilde{f}(x_1, x_2) = \left[\tilde{f}_1(x_1, x_2)\ \tilde{f}_2(x_1, x_2)\right]$. Definitions of intermediate variables $v_{-1}, \ldots, v_7$ can be found in table 3.1 or table 3.2

Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	v'_{-1}	$= x'_1$	$= 1$
v_0	$= x_2$	$= 3$	v'_0	$= x'_2$	$= 0$
v_1	$= \cos v_{-1}$	$= \cos 2$	v'_1	$= -v'_{-1} \sin v_{-1}$	$= -1 \cdot \sin 2$
v_2	$= v_{-1} v_0$	$= 2 \cdot 3$	v'_2	$= v'_{-1} v_0 + v_{-1} v'_0$	$= 1 \cdot 3 + 2 \cdot 0$
v_3	$= v_0^3$	$= 3^3$	v'_3	$= 3 v_o^2 v'_0$	$= 3 \cdot 2^2 \cdot 0$
v_4	$= \ln v_{-1}$	$= \ln 2$	v'_4	$= v'_{-1} / v_{-1}$	$= 1/2$
v_5	$= v_2 + v_1$	$= 6 - 0.416$	v'_5	$= v'_2 + v'_1$	$= 3 - 0.909$
v_6	$= v_3 + v_4$	$= 27 + 0.693$	v'_6	$= v'_3 + v'_4$	$= 0 + 0.5$
v_7	$= v_6 - v_0$	$= 27.693 - 3$	v'_7	$= v'_6 - v'_0$	$= 0.5 - 0$
y_1	$= v_5$	$= 5.584$	$\mathbf{y'_1}$	$= \mathbf{v'_5}$	$= \mathbf{2.091}$
y_2	$= v_7$	$= 24.693$	$\mathbf{y'_2}$	$= \mathbf{v'_7}$	$= \mathbf{0.5}$

Table 3.1: Forward mode AD example to evaluate the derivatives $\frac{\partial y_1}{\partial x_1}$ and $\frac{\partial y_2}{\partial x_1}$ of $\tilde{f}(x_1, x_2)$ at $[x_1, x_2] = [2, 3]$. x'_1 and x'_2 are respectively set to 1 and 0 in order to derive only respect x_1. On the left is reported the forward evaluation trace, on the right the tangent one

3.1.1 Forward mode

In order to compute the derivative of the function $\tilde{f}$, reported in (3.1.1), respect to x_1 we start by considering the evaluation trace on the left-hand side of table 3.1 and we associate to each intermediate variable v_i the derivative:

$$v_i' = \frac{\partial v_i}{\partial x_1}$$

Here the variable with respect to which we differentiate remain the same independently on i. Moving from top to bottom in the forward primal trace the corresponding tangent (derivative) trace, presented on the right-hand side of table 3.1, is generated by applying the chain rule to each encountered operation. After the primals v_i are evaluated, also the corresponding tangents v_i' are, again, from top to bottom, which means that tangent trace can be computed in parallel with the forward trace. This gives us the desired derivatives in the final variables $v_5' = \frac{\partial y_1}{\partial x_1}$ and $v_7' = \frac{\partial y_2}{\partial x_1}$.

Forward mode can also be employed to evaluate the Jacobian of a generic function $f\colon \mathbb{R}^n \to \mathbb{R}^m$ at a point $\boldsymbol{x} = \boldsymbol{a}$ by performing n distinct forward passes. By setting only one variable $x_i' = 1$ and the others to zero we obtain:

$$y_j' = \left.\frac{\partial y_j}{\partial x_i}\right|_{\boldsymbol{x}=\boldsymbol{a}} \qquad j = 1, \dots, m.$$

and reiterating for $i = 1, \dots, n$, placing the resulting vectors side by side, we eventually obtain the desired Jacobian:

$$\boldsymbol{J}_f = \left.\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}\right|_{\boldsymbol{x}=\boldsymbol{a}}$$

Furthermore, by properly fine-tuning the values of the variables $x_1', \dots, x_n'$, it is possible to compute efficiently and in a matrix-free way Jacobian-vector products:

$$\boldsymbol{J}_f \boldsymbol{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix}$$

To accomplish this all that needs to be done is simply initializing $\boldsymbol{x}' = \left[x_1', \dots, x_n'\right]$ with $\boldsymbol{r}$; this result is particularly important in the evaluation of directional derivatives. Before moving on, it is crucial to point out that forward mode AD:

- for a function $f\colon \mathbb{R} \to \mathbb{R}^m$ allows to evaluate all its derivatives in just one forward pass, regardless of m;
- for a scalar field $f\colon \mathbb{R}^n \to \mathbb{R}$, the evaluation of its gradient $\nabla f = \left[\frac{\partial y}{\partial x_1}, \ldots, \frac{\partial y}{\partial x_n}\right]$ always require n evaluations (since gradient is nothing more than a Jacobian of size $1 \times n$).

This means that the here explained implementation of Automatic Differentiation has a computational cost which scales linearly with the number of function inputs. It is effective during the computation of derivative for cases $f\colon \mathbb{R}^n \to \mathbb{R}^m$ where $n \ll m$; for cases $n \gg m$ *reverse mode* is more beneficial: we will discover why in the next subsection.

3.1.2 Reverse mode

In this case, as opposed to forward mode AD, the derivatives are propagated back from a given output. A demonstration of how it works is presented in table 3.2 where we compute the sensitivities of output $y_1 = \tilde{f}_1(x_1, x_2)$ of $\tilde{f}$ respect the inputs x_1 and x_2. Different variables, called adjoints, are associated to each variable v_i of the Forward Primal Trace and each of them is defined as:

$$\overline{v}_i = \frac{\partial y_1}{\partial v_i}$$

An adjoint variable is nothing more than the sensitivity of the output y_1 with respect to changes in v_i. It is important for the reader to note that, with this mode, the variable with respect to the derivatives are computed is not fixed as in forward mode AD.

Forward Primal Trace		
v_{-1}	$= x_1$	$= 2$
v_0	$= x_2$	$= 3$
v_1	$= \cos v_{-1}$	$= \cos 2$
v_2	$= v_{-1} v_0$	$= 2 \cdot 3$
v_3	$= v_0^3$	$= 3^3$
v_4	$= \ln v_{-1}$	$= \ln 2$
v_5	$= v_2 + v_1$	$= 6 - 0.416$
v_6	$= v_3 + v_4$	$= 27 + 0.693$
v_7	$= v_6 - v_0$	$= 27.693 - 3$
y_1	$= v_5$	$= 5.584$
y_2	$= v_7$	$= 24.693$

Reverse Adjoint (Derivative) Trace			
$\overline{\mathbf{x}}_\mathbf{1}$	$= \frac{\partial \mathbf{y_1}}{\partial \mathbf{v_{-1}}} \frac{\partial \mathbf{v_{-1}}}{\partial \mathbf{x_1}}$	$= \overline{\mathbf{v}}_\mathbf{-1} \cdot \mathbf{1}$	$= 2.091$
$\overline{\mathbf{x}}_\mathbf{2}$	$= \frac{\partial \mathbf{y_1}}{\partial \mathbf{v_0}} \frac{\partial \mathbf{v_0}}{\partial \mathbf{x_2}}$	$= \overline{\mathbf{v}}_\mathbf{0} \cdot \mathbf{1}$	$= 2$
$\overline{v}_{-1}$	$= \frac{\partial y_1}{\partial v_1} \frac{\partial v_1}{\partial v_{-1}} + \frac{\partial y_1}{\partial v_2} \frac{\partial v_2}{\partial v_{-1}} + \frac{\partial y_1}{\partial v_4} \frac{\partial v_4}{\partial v_{-1}}$	$= -\overline{v}_1 \sin(v_{-1}) + \overline{v}_2 v_0 + \overline{v}_4 / v_{-1}$	$= 2.091$
$\overline{v}_0$	$= \frac{\partial y_1}{\partial v_2} \frac{\partial v_2}{\partial v_0} + \frac{\partial y_1}{\partial v_3} \frac{\partial v_3}{\partial v_0}$	$= \overline{v}_2 v_{-1} + 3\overline{v}_3 v_0^2$	$= 2$
$\overline{v}_1$	$= \frac{\partial y_1}{\partial v_5} \frac{\partial v_5}{\partial v_1}$	$= \overline{v}_5 \cdot 1$	$= 1$
$\overline{v}_2$	$= \frac{\partial y_1}{\partial v_5} \frac{\partial v_5}{\partial v_2}$	$= \overline{v}_5 \cdot 1$	$= 1$
$\overline{v}_3$	$= \frac{\partial y_1}{\partial v_6} \frac{\partial v_6}{\partial v_3}$	$= \overline{v}_6 \cdot 1$	$= 0$
$\overline{v}_4$	$= \frac{\partial y_1}{\partial v_6} \frac{\partial v_6}{\partial v_4}$	$= \overline{v}_6 \cdot 1$	$= 0$
$\overline{v}_6$	$= \frac{\partial y_1}{\partial v_7} \frac{\partial v_7}{\partial v_6}$	$= \overline{v}_7 \cdot 1$	$= 0$
$\overline{v}_5$	$= \overline{y}_1$		$= 1$
$\overline{v}_7$	$= \overline{y}_2$		$= 0$

Table 3.2: Reverse mode AD example with $[y_1, y_2] = \tilde{f}(x_1, x_2)$ evaluated at $[x_1, x_2] = [2, 3]$. First, primal trace is evaluated (table above) and then adjoint variables are computed, from the bottom up, in a second phase (table below). The initialization $\overline{v}_5 = \frac{\partial y_1}{\partial v_5} = \frac{\partial y_1}{\partial y_1} = \overline{y}_1 = 1$ and $\overline{v}_7 = \frac{\partial y_1}{\partial v_7} = \frac{\partial y_1}{\partial y_2} = \overline{y}_2 = 0$ is such that it allows the sensitivities to be calculated with respect the fist output

Since typical usage of chain rule is forward derivatives propagation, reverse mode AD might appear confusing at first sight; to make it clearer we show how the chain rule can be used to back-propagating derivatives in order to compute the contribution $\overline{v}_0$ of the change in variable v_0 to the change in the output y_1. From figure 3.1 can be seen that the only way variable v_0 can affect y_1 is through affecting v_2 and v_3, so its contribution to the change in y_1 can be computed using the chain rule as follow:

$$\frac{\partial y_1}{\partial v_0} = \frac{\partial y_1}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y_1}{\partial v_3}\frac{\partial v_3}{\partial v_0} \tag{3.1.2}$$

which can be rewritten in

$$\begin{aligned} \frac{\partial y_1}{\partial v_0} &= \overline{v}_2\frac{\partial v_2}{\partial v_0} + \overline{v}_3\frac{\partial v_3}{\partial v_0} \\ &= \overline{v}_2 v_{-1} + 3\overline{v}_3 v_0^2 \end{aligned} \tag{3.1.3}$$

where quantities $\overline{v}_2$ and $\overline{v}_3$ are known from previous passages and derivatives $\frac{\partial v_2}{\partial v_0}$, $\frac{\partial v_3}{\partial v_0}$, can be easily computed by evaluating the result of symbolic differentiation of the elementary operations on the primals v_i.

In light of the previous example, it is clear that, to compute derivatives, primals v_i must be known along their dependencies within the computational graph. To do so two phases are required:

- A *forward step* where variables v_i are populated and their dependencies recorded;
- A *reverse step* where adjoints $\overline{v}_i$ are evaluated from outputs to inputs using values obtained in the first step.

We remark that once the backward pass is completed we get *all* the sensitivities $\frac{\partial y_j}{\partial x_1}, \ldots, \frac{\partial y_j}{\partial x_n}$ in one single pass; this means that for scalar fields $f\colon \mathbb{R}^n \to \mathbb{R}$ the computation of the gradient ∇f require only one application of reverse mode as opposed to the n passes required in case of forward mode. Nevertheless, this advantage comes at the cost of increased memory requirements which grows proportionally (in the worst case) to the number of operations in the evaluated function [28].

Generalizing the reasoning to functions $f\colon \mathbb{R}^n \to \mathbb{R}^m$, similarly to how it was done in previous section, we can conclude that reverse mode AD can also be used to evaluate Jacobians at a point; in particular, since each pass of reverse AD allows to compute one row of the Jacobian, its usage is advantageous when $m \ll n$. With regard to products between Jacobain and vector, reverse AD allows to compute

efficiently vector-Jacobian products of type:

$$\boldsymbol{r}^T \boldsymbol{J}_f = \begin{bmatrix} r_1 & \dots & r_m \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \tag{3.1.4}$$

by initializing $\overline{\boldsymbol{y}} = \begin{bmatrix} \overline{y}_1, \dots, \overline{y}_m \end{bmatrix}$ with $\boldsymbol{r}$.

3.2 Application to design optimization

Design optimization is the process of finding the best design parameters $\boldsymbol{l} = [l_1, \dots, l_N]$ that satisfy project requirements [29]. Requirements and objectives, which are usually multiple and conflicting, are typically expressed in the form of a scalar function $J \colon \mathbb{R}^M \times \mathbb{R}^N \to \mathbb{R}$ named *cost* or *objective function*. J depends on the state $\boldsymbol{u} \in \mathbb{R}^M$ of the problem at hand and the design parameters $\boldsymbol{l} \in \mathbb{R}^N$; $\boldsymbol{u}$ is generally found as solution of a system of linear equations $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$, which arise from the imposition of the constraints of the problem, where $\boldsymbol{A} \in \mathbb{R}^{M \times M}$ and $\boldsymbol{b} \in \mathbb{R}^M$ depends on some way on $\boldsymbol{l}$. In figure 3.2 can be found a scheme of the whole process.

This type of process is of interest since it is encountered in the most diverse fields, from computer science, where for example the proper weights of a neural network has to be found in order to minimize the error between its predicted and the desired output respect a given set of inputs, to mechanical engineering where, for example, one may want to modify the shape of an heat sink to maximize the amount of the heat transferred to the environment.

To find the best parameters for a given project a variety of iterative approaches can be pursued during the optimization phase:

Manual approach where each parameter $l_1, \dots, l_N$ is adjusted one at a time. Unfortunately doing so tends to leads to suboptimal results;

Brute-force approach where all possible combinations of design parameters are evaluated. However, this is time-expensive and for large-scale projects, where parameters can be hundreds, thousands, or even more, optimal parameters may not be found in reasonable time-frames;

Gradient approach where the gradient $\frac{dJ}{d\boldsymbol{l}}$ permits both to update all the parameters $l_1, \dots, l_N$ at the same time and to disregard most of the parameter values that are sub-optimal by considering only those suggested by its direction.

For the effectiveness of the last mentioned approach, is crucial to ensure an efficient computation of the gradient $\frac{dJ}{d\boldsymbol{l}}$ which must require as few steps as possible and be

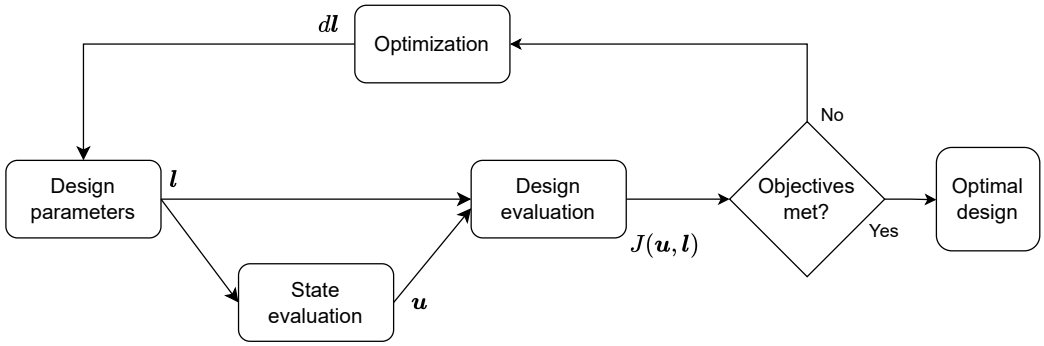


Figure 3.2: Scheme of a design optimization process. From an initial design defined by a set of parameters $\boldsymbol{l}$ the associated state $\boldsymbol{u}$ is derived. The state in conjunction with the parameters are then used to evaluate the design through a function J. If the objectives are met the design is kept otherwise the whole procedure is repeated with a new set of parameters obtained by modifying the previous ones by a quantity $d\boldsymbol{l}$

efficient even for problems involving huge number of design parameters. To meet these requirements Automatic Differentiation (AD) in its reverse-mode declination is necessary.

Given its advantages in design optimization we have learned that using the gradient is a good course of action. To directly evaluate $\frac{dJ}{d\boldsymbol{l}}$, in case of a particular set of design parameters defined by $\boldsymbol{l}$ at a given state $\boldsymbol{u}$, we would compute:

$$\frac{dJ}{d\boldsymbol{l}}^T = \frac{\partial J}{\partial \boldsymbol{u}}^T \frac{d\boldsymbol{u}}{d\boldsymbol{l}} + \frac{\partial J}{\partial \boldsymbol{l}}^T \tag{3.2.1}$$

where notations $\frac{d}{dx}$ and $\frac{\partial}{\partial x}$ indicate the total and the partial derivatives respect the variable x respectively. Terms $\frac{\partial J}{\partial \boldsymbol{u}}$ and $\frac{\partial J}{\partial \boldsymbol{l}}$ can be efficiently obtained by means of reverse-mode AD, as explained in subsection **??** on page ??, since J is known analytically. The matrix $\frac{d\boldsymbol{u}}{d\boldsymbol{l}}$, on the other hand, require more careful considerations.

Differentiating the problem constraints $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$ with respect to the parameters $\boldsymbol{l}$ gives:

$$\frac{d\boldsymbol{A}}{d\boldsymbol{l}}\boldsymbol{u} + \boldsymbol{A}\frac{d\boldsymbol{u}}{d\boldsymbol{l}} = \frac{d\boldsymbol{b}}{d\boldsymbol{l}} \tag{3.2.2}$$

which allows to obtain the sought term $\frac{d\boldsymbol{u}}{d\boldsymbol{l}}$ as:

$$\frac{d\boldsymbol{u}}{d\boldsymbol{l}} = \boldsymbol{A}^{-1}\left(\frac{d\boldsymbol{b}}{d\boldsymbol{l}} - \frac{d\boldsymbol{A}}{d\boldsymbol{l}}\boldsymbol{u}\right) \tag{3.2.3}$$

We notice that $\frac{d\boldsymbol{u}}{d\boldsymbol{l}}$ is a matrix of size $M \times N$ whose j-th column represent the sensitivity of the state $\boldsymbol{u}$ respect to the parameter l_j. In practice it is populated

column-wise by solving equation (3.2.3) N times when the derivative is taken with respect the j-th design parameter l_j rather than $\boldsymbol{l}$: this gives its j-th column. In this process $\frac{d\boldsymbol{b}}{d\boldsymbol{l}}$ and $\frac{d\boldsymbol{A}}{d\boldsymbol{l}}$ terms can be simply obtained using AD, if $\boldsymbol{b}$ and $\boldsymbol{A}$ are known analytically, and once $\frac{d\boldsymbol{u}}{d\boldsymbol{l}}$ is computed it can be substituted in equation (3.2.1) to obtain the gradient which symbolically reads as:

$$\frac{dJ}{d\boldsymbol{l}} = \frac{\partial J}{\partial \boldsymbol{u}}^T \left[\boldsymbol{A}^{-1} \left(\frac{d\boldsymbol{b}}{d\boldsymbol{l}} - \frac{d\boldsymbol{A}}{d\boldsymbol{l}} \boldsymbol{u} \right) \right] + \frac{\partial J}{\partial \boldsymbol{l}}^T \tag{3.2.4}$$

However the computational costs of this naive procedure scales linearly with the number of the design parameters N, since N solutions of liner systems are required in order to construct $\frac{d\boldsymbol{u}}{d\boldsymbol{l}}$. We finally remark that each of these inversions has the same computational cost of solving $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$ for $\boldsymbol{u}$ and if the solution represent the output of a particularly time consuming computation, as a CFD simulation is an example of, this is particularly penalizing. A more efficient gradient computation is therefore required.

To do so we can employ the *adjoint method* which simply consist of a smarter bracketing of the equation which gives the gradient. In fact is possible to rewrite (3.2.4) as:

$$\begin{aligned} \frac{dJ}{d\boldsymbol{l}} &= \left[\frac{\partial J}{\partial \boldsymbol{u}}^T \boldsymbol{A}^{-1} \right] \left(\frac{d\boldsymbol{b}}{d\boldsymbol{l}} - \frac{d\boldsymbol{A}}{d\boldsymbol{l}} \boldsymbol{u} \right) + \frac{\partial J}{\partial \boldsymbol{l}}^T \\ &= \boldsymbol{\lambda}^T \left(\frac{d\boldsymbol{b}}{d\boldsymbol{l}} - \frac{d\boldsymbol{A}}{d\boldsymbol{l}} \boldsymbol{u} \right) + \frac{\partial J}{\partial \boldsymbol{l}}^T \end{aligned} \tag{3.2.5}$$

where the vector $\boldsymbol{\lambda} \in \mathbb{R}^L$, whose elements are called *adjoint variables*, can be found by solving:

$$\boldsymbol{A}^T \boldsymbol{\lambda} = \frac{\partial J}{\partial \boldsymbol{u}} \tag{3.2.6}$$

By doing so is possible to solve the system in equation (3.2.6), which is called *adjoint problem*, only *once* independently on the number N of design parameters for each gradient computation: this significantly reduces the computational burden of the whole optimization process. Bear in mind that the adjoint problem has the same size as the problem defined by the constraints $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$ and the computational cost for their solution is the same.

At this point it is worth noting that the adjoint method explained above is just a particular application of reverse mode Automatic Differentiation (AD). To make the connections between the two methods clearer we note that the first computation of the objective function's gradient, presented in equation (3.2.1), contains the following vector-Jacobian product:

$$\frac{\partial J}{\partial \boldsymbol{u}}^T \frac{d\boldsymbol{u}}{d\boldsymbol{l}} \tag{3.2.7}$$

where $\frac{\partial J}{\partial \boldsymbol{u}}^T \in \mathbb{R}^M$ is the vector that multiply $\frac{d\boldsymbol{u}}{d\boldsymbol{l}} \in \mathbb{R}^{M \times N}$, the Jacobian of the state $\boldsymbol{u}$ respect the design parameters $\boldsymbol{l}$. From subsection 3.1.2 on page 28 we known that AD allows to compute the resulting vector without explicitly computing the Jacobian $\frac{d\boldsymbol{u}}{d\boldsymbol{l}}$. But this is precisely what the adjoint method does through equations (3.2.1 - 3.2.6) by first applying the chain rule to the leftmost term in the gradient formula and then parenthesizing the result in order to make the number of inversions of the matrix $\boldsymbol{A}$ independent on the number of parameters. The result of the whole process is then summarized by:

$$\boldsymbol{\lambda}^T \left(\frac{d\boldsymbol{b}}{d\boldsymbol{l}} - \frac{d\boldsymbol{A}}{d\boldsymbol{l}} \boldsymbol{u} \right) \tag{3.2.8}$$

where $\boldsymbol{\lambda}$ solves equation (3.2.6) and the Jacobian $\frac{d\boldsymbol{u}}{d\boldsymbol{l}}$ is no more present. Furthermore the two-steps process of the reverse mode AD for the gradient computation is still there even in the adjoint method:

- the forward pass, used for the computation of variables v_i in table 3.2, now is the single solution of the system $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$ in order to find the state $\boldsymbol{u}$ used in term (3.2.8);
- the reverse pass, done in order to obtain the adjoint variables $\overline{v}_i$ in table 3.2, is now equivalent to the solution of $\boldsymbol{A}^T\boldsymbol{\lambda} = \frac{\partial J}{\partial \boldsymbol{u}}$ (we would like to reiterate again that reverse and forward pass has the *same* computational complexity).

After this concise overview of the adjoint method applied to generic design optimization problems, we now proceed to the next section where we examine in more detail its application to those problems which state $\boldsymbol{u}$ is obtained solving a Partial Differential Equation by means of RBF-FD methods.

3.3 Application to RBF-FD

In section 2.6 we have seen how the RBF-FD solver for Partial Differential Equations (PDEs) is implemented. Computational Fluid Dynamics (CFD), where PDEs are employed to model fluid flow and heat transfer problems, is one of the main areas where the application of the RBF-FD method could bring great benefits.

Once a generic description for the design optimization problem has been written it has to be translated into a mathematical statement in order to solve it through an optimization algorithm. To do so the following items are used for a correct formulation:

- *Design variables* represented by a vector $\boldsymbol{l} = [l_1, \ldots, l_L] \in \mathbb{R}^L$. We consider only continuous variables since they control the shape objects which are allowed to vary continuously in the space;

- *Objective function* indicated with $J\colon \mathbb{R}^{N_I} \times \mathbb{R}^L \to \mathbb{R}$. It is a scalar function that yields a quantity used to determines if one design is better than another. It is an explicit function of the state $\boldsymbol{u}_I$ of the problem, which is obtained from the RBF-FD solver, and the design parameters $\boldsymbol{l}$. Its choice is crucial: it has to represent the real goal of the whole optimization. A poor choice would lead to a mathematical optimum which does not match with the engineering optimum;

- *Constraints* represented by the RBF-FD governing equations: these are reported in equation (2.6.18) in subsection 2.6.2. We reiterate them here, in a more compact form, since they will be used extensively throughout the rest of this section:

$$\boldsymbol{C}_I \boldsymbol{u}_I + \boldsymbol{C}_B \boldsymbol{u}_B = \boldsymbol{f} \tag{3.3.1}$$

  where we have used the following notation: $\boldsymbol{u}_I = [u(\boldsymbol{x}_1), \dots u(\boldsymbol{x}_{N_I})]$, $\boldsymbol{u}_B = [g(\boldsymbol{x}_{N_I+1}), \dots, g(\boldsymbol{x}_N)]$ and $\boldsymbol{f} = [f(\boldsymbol{x}_1), \dots, f(\boldsymbol{x}_{N_I})]$, recalling that N is the number of RBF-FD nodes scattered throughout the physical domain $\Omega \cup \partial\Omega$ of the system wich is represented by the volume of the object. The first N_I nodes are contained within it, the remaining ones are placed on its surface. Before proceeding is worth noting that, in general, matrices $\boldsymbol{C}_I$ and $\boldsymbol{C}_B$, and vectors $\boldsymbol{f}$ and $\boldsymbol{u}_B$ depend on design variables $\boldsymbol{l}$. Therefore the solution $\boldsymbol{u}_I$ will also be a function (indirectly through constraint (3.3.1)) of design variables.

Therefore the resulting optimization problem can be written as:

$$\begin{aligned} \text{minimize} \quad & J(\boldsymbol{u}_I, \boldsymbol{l}) \\ \text{by varying} \quad & \boldsymbol{u}_I \\ & \boldsymbol{l} \\ \text{subject to} \quad & \boldsymbol{C}_I \boldsymbol{u}_I + \boldsymbol{C}_B \boldsymbol{u}_B = \boldsymbol{f} \end{aligned} \tag{3.3.2}$$

which optimum is found through the gradient approach explained in section 3.2. In general with a similar formulation can be solved also problems which require the maximization of the objective function since $\max[J(\boldsymbol{u}_I, \boldsymbol{l})] = -\min[-J(\boldsymbol{u}_I, \boldsymbol{l})]$. Finally we remark that the variables that can be modified at each optimization cycle are only the design variables $\boldsymbol{l}$ since the result of CFD simulation $\boldsymbol{u}_I$ is fully determined from constraints (3.3.1);

In the two following subsections we will explain how the gradient $\frac{dJ}{d\boldsymbol{l}}$ can be computed efficiently by means of the adjoint method for the optimization problem (3.3.2) in case of one and three-dimensional objects.

3.3.1 1D

In this case we consider a general cost function $J\colon \mathbb{R}^{N_I \times L} \to \mathbb{R}$. Thanks to this generality its sensitivities respect the design parameters are given by the same exact formula used in general design optimization problems:

$$\frac{dJ}{d\boldsymbol{l}}^T = \frac{\partial J}{\partial \boldsymbol{u}_I}^T \frac{d\boldsymbol{u}_I}{d\boldsymbol{l}} + \frac{\partial J}{\partial \boldsymbol{l}}^T \tag{3.3.3}$$

Once more, what is now lacking is the term $\frac{d\boldsymbol{u}_I}{d\boldsymbol{l}}$. The sensitivities of the state respect the parameters can be computed by differentiating equation (3.3.1) which gives:

$$\frac{d\boldsymbol{C}_I}{d\boldsymbol{l}}\boldsymbol{u}_I + \boldsymbol{C}_I\frac{d\boldsymbol{u}_I}{d\boldsymbol{l}} + \frac{d\boldsymbol{C}_B}{d\boldsymbol{l}}\boldsymbol{u}_B + \boldsymbol{C}_B\frac{d\boldsymbol{u}_B}{d\boldsymbol{l}} = \frac{d\boldsymbol{f}}{d\boldsymbol{l}} \tag{3.3.4}$$

Before proceeding a note on the matrix $\frac{d\boldsymbol{u}_B}{d\boldsymbol{l}}$ could be useful: one might think that it should not be present since values in $\boldsymbol{u}_B$ are fixed from boundary conditions of the problem (2.6.5) (i.e. they are not dependent on the design parameters). The previous statements holds true only for Dirichlet Boundary Conditions (BCs), but not in general since in case of Neumann and Robin BCs the values of $\boldsymbol{u}_B$ depends also on their normal derivatives respect to the surface $\partial\Omega$ whose shape depends on $\boldsymbol{l}$. This simply means that only the rows of $\frac{d\boldsymbol{u}_B}{d\boldsymbol{l}}$ associated to Dirichlet points will make up of zeros elements.

Once $\frac{d\boldsymbol{u}_I}{d\boldsymbol{l}}$ is isolated from the previous equation it can be plugged in (3.3.3) which in turns become:

$$\begin{aligned}\frac{dJ}{d\boldsymbol{l}}^T &= \frac{\partial J}{\partial \boldsymbol{u}_I}^T\left[\boldsymbol{C}_I^{-1}\left(\frac{d\boldsymbol{f}}{d\boldsymbol{l}} - \frac{d\boldsymbol{C}_I}{d\boldsymbol{l}}\boldsymbol{u}_I - \frac{d\boldsymbol{C}_B}{d\boldsymbol{l}}\boldsymbol{u}_B - \boldsymbol{C}_B\frac{d\boldsymbol{u}_B}{d\boldsymbol{l}}\right)\right] + \frac{\partial J}{\partial \boldsymbol{l}}^T \\ &= \frac{\partial J}{\partial \boldsymbol{u}_I}^T\left[\boldsymbol{C}_I^{-1}\left(\frac{d\boldsymbol{f}}{d\boldsymbol{l}} - \frac{d\boldsymbol{C}}{d\boldsymbol{l}}\boldsymbol{u} - \boldsymbol{C}_B\frac{d\boldsymbol{u}_B}{d\boldsymbol{l}}\right)\right] + \frac{\partial J}{\partial \boldsymbol{l}}^T\end{aligned} \tag{3.3.5}$$

where $\frac{d\boldsymbol{C}}{d\boldsymbol{l}} \in \mathbb{R}^{N_I \times N}$ is the matrix resulting from the concatenation of the rows of $\frac{d\boldsymbol{C}_I}{d\boldsymbol{l}} \in \mathbb{R}^{N_I \times N_I}$ and $\frac{d\boldsymbol{C}_B}{d\boldsymbol{l}} \in \mathbb{R}^{N_B \times N_B}$, and $\boldsymbol{u} \in \mathbb{R}^N$ is the vector given by the concatenation of the elements of $\boldsymbol{u}_I \in \mathbb{R}^{N_I}$ and $\boldsymbol{u}_B \in \mathbb{R}^{N_B}$. Now the adjoint method can be employed in order to avoid performing the inversion of matrix $\boldsymbol{C}_I$ once for each parameter in $\boldsymbol{l}$, yielding:

$$\frac{dJ}{d\boldsymbol{l}}^T = \boldsymbol{\lambda}_1^T\left(\frac{d\boldsymbol{f}}{d\boldsymbol{l}} - \frac{d\boldsymbol{C}}{d\boldsymbol{l}}\boldsymbol{u} - \boldsymbol{C}_B\frac{d\boldsymbol{u}_B}{d\boldsymbol{l}}\right) + \frac{\partial J}{\partial \boldsymbol{l}}^T \tag{3.3.6}$$

where $\boldsymbol{\lambda}_1$ is found by solving *once*:

$$\boldsymbol{C}_I^T\boldsymbol{\lambda}_1 = \frac{\partial J}{\partial \boldsymbol{u}_I} \tag{3.3.7}$$

The unknown terms involving derivatives computation on the right-hand side can be computed easily through AD except for $\frac{d\boldsymbol{C}}{d\boldsymbol{l}}\boldsymbol{u}$. This terms deserves more attentions since requires the differentiation of the global matrix $\boldsymbol{C}$ which is obtained by solving N_I local systems, each associated with a different stencil $\mathcal{X}_i$.

In order to examine it more closely we can define a matrix $\boldsymbol{Q} \in \mathbb{R}^{N_I \times L}$ as follows:

$$\boldsymbol{Q} = \frac{d\boldsymbol{C}}{d\boldsymbol{l}}\boldsymbol{u} \tag{3.3.8}$$

whose elements $q_{i,j}$ are given by:

$$q_{i,j} = \frac{d\boldsymbol{C}_{[i,:]}}{dl_j}\boldsymbol{u} \tag{3.3.9}$$

where notation $\boldsymbol{C}_{[i,:]}$ is used to indicate the i-th row of the matrix $\boldsymbol{C}$. To obtain $\frac{d\boldsymbol{C}}{d\boldsymbol{l}}\boldsymbol{u}$ is thus sufficient to compute each element of $\boldsymbol{Q}$ through equation (3.3.9). However, before doing so, it is worth rewriting the equation for $q_{i,j}$ more explicitly.

Recall that elements that make up the i-th row of $\boldsymbol{C}$ are found by solving the local system (2.6.15):

$$\boldsymbol{M}_{BC}^T \begin{bmatrix} \boldsymbol{c}_I(\boldsymbol{x}_i) \\ \boldsymbol{c}_B(\boldsymbol{x}_i) \\ \boldsymbol{c}_p(\boldsymbol{x}_i) \end{bmatrix} = \begin{bmatrix} \mathcal{L}\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i) \end{bmatrix} \tag{3.3.10}$$

in case of RBF-FD method, or the local system (2.7.20):

$$\boldsymbol{M}_{BC}^T \begin{bmatrix} \boldsymbol{c}_I(\boldsymbol{x}_i) \\ \boldsymbol{c}_B(\boldsymbol{x}_i) \\ \boldsymbol{c}_p(\boldsymbol{x}_i) \end{bmatrix} = \begin{bmatrix} \mathcal{L}_1\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,I}) \\ \mathcal{L}_1\mathcal{B}_2\boldsymbol{\Phi}(\boldsymbol{x}_i, \mathcal{X}_{i,B}) \\ \mathcal{L}\boldsymbol{p}(\boldsymbol{x}_i) \end{bmatrix} \tag{3.3.11}$$

in case of RBF-HFD method. The key aspect that both cases share is that the aforementioned equations arises from the information contained in a single stencil $\mathcal{X}_i$ and, as explained in subsection 2.6.2, only the first m elements of $\boldsymbol{c}(\boldsymbol{x}_i) = \left[\boldsymbol{c}_I(\boldsymbol{x}_i), \boldsymbol{c}_B(\boldsymbol{x}_i), \boldsymbol{c}_p(\boldsymbol{x}_i)\right]$ form the i-th row of $\boldsymbol{C}$. Furthermore, since $m \ll N$, row $\boldsymbol{C}_{[i,:]}$ is sparse; which in turns implies that not all the elements of $\boldsymbol{u}$ are involved in the computation of $q_{i,j}$.

By letting $\tilde{\boldsymbol{u}}_i \in \mathbb{R}^m$ the vector composed by the components of $\boldsymbol{u}$ associated to the non zero elements of $\boldsymbol{C}_{[i,:]}$, we can rewrite the elements of $\boldsymbol{Q}$ reported in (3.3.9) as:

$$q_{i,j} = \frac{d\boldsymbol{c}(\boldsymbol{x}_i)^T}{dl_j} \begin{bmatrix} \tilde{\boldsymbol{u}}_i \\ \boldsymbol{0} \end{bmatrix} \tag{3.3.12}$$

where the zero-vector concatenated to $\tilde{\boldsymbol{u}}_i$ has the same length of $\boldsymbol{c}_p(\boldsymbol{x}_i)$. If we now differentiate equation (3.3.11) respect the j-th design parameter, we obtain:

$$\frac{d\boldsymbol{M}_{BC}^T}{dl_j}\boldsymbol{c}(\boldsymbol{x}_i) + \boldsymbol{M}_{BC}^T\frac{d\boldsymbol{c}(\boldsymbol{x}_i)}{dl_j} = \frac{d\boldsymbol{h}}{dl_j} \tag{3.3.13}$$

where the vector $\boldsymbol{h} \in \mathbb{R}^{m+M}$ is used as a shorthand for the vector present on the right-hand side of equation (3.3.11). Now from the last equation we can isolate:

$$\frac{d\boldsymbol{c}(\boldsymbol{x}_i)^T}{dl_j} = \left(\frac{d\boldsymbol{h}}{dl_j} - \frac{d\boldsymbol{M}_{BC}^T}{dl_j}\boldsymbol{c}(\boldsymbol{x}_i)\right)^T \boldsymbol{M}_{BC}^{-1} \tag{3.3.14}$$

which once plugged in (3.3.12) allows to rewrite $q_{i,j}$ as:

$$q_{i,j} = \left[\left(\frac{d\boldsymbol{h}}{dl_j} - \frac{d\boldsymbol{M}_{BC}^T}{dl_j}\boldsymbol{c}(\boldsymbol{x}_i)\right)^T \boldsymbol{M}_{BC}^{-1}\right] \begin{bmatrix}\tilde{\boldsymbol{u}}_i \\ \boldsymbol{0}\end{bmatrix} \tag{3.3.15}$$

But now we can clearly see that in order to populate the matrix $\boldsymbol{Q}$ we should invert $N_I L$ matrices $\boldsymbol{M}_{BC}$ of size $(m+M)\times(m+M)$ since we need to compute $\frac{d\boldsymbol{c}(\boldsymbol{x}_i)^T}{dl_j}$. These matrices are smaller than $\boldsymbol{C}_I$, but in any case their inversion is still problematic, since both the number of parameters and the number of nodes are huge. The problem, here, is very similar to the one related to design optimization in section 3.2 where there was a linear relationship between the number of matrix inversions and the number of design parameters: consequently, even here, to improve the situation it is possible to rely on the adjoint method. Modifying the parentheses of equation (3.3.15) it is possible to write:

$$\begin{aligned} q_{i,j} &= \left(\frac{d\boldsymbol{h}}{dl_j} - \frac{d\boldsymbol{M}_{BC}^T}{dl_j}\boldsymbol{c}(\boldsymbol{x}_i)\right)^T \left(\boldsymbol{M}_{BC}^{-1}\begin{bmatrix}\tilde{\boldsymbol{u}}_i \\ \boldsymbol{0}\end{bmatrix}\right) \\ &= \left(\frac{d\boldsymbol{h}}{dl_j} - \frac{d\boldsymbol{M}_{BC}^T}{dl_j}\boldsymbol{c}(\boldsymbol{x}_i)\right)^T \boldsymbol{\lambda}_{2,i} \end{aligned} \tag{3.3.16}$$

where the adjoint vector $\boldsymbol{\lambda}_{2,i} \in \mathbb{R}^{m+M}$ is found as solution of:

$$\boldsymbol{M}_{BC}\boldsymbol{\lambda}_{2,i} = \begin{bmatrix}\tilde{\boldsymbol{u}}_i \\ \boldsymbol{0}\end{bmatrix} \tag{3.3.17}$$

and has to be computed once for each row of $\boldsymbol{Q}$ (or equivalently $\frac{d\boldsymbol{C}}{dl}\boldsymbol{u}$). The crucial point is that this second adjoint implementation allows to make the computational cost to obtain $\frac{d\boldsymbol{C}}{dl}\boldsymbol{u}$ independent of the number of design parameter L: the number of systems (3.3.17) that has to be solved, whose cost is similar to the inversion of $\boldsymbol{M}_{BC}$, is now always N_I.

In light of what we have been observed we are now able to compute the whole gradient $\frac{dJ}{d\boldsymbol{l}}^T$ with a computation effort analogous to two RBF-HFD:

1. a first vanilla application of RBF-HFD is required in order to compute the terms $\boldsymbol{u}$, $\boldsymbol{C}_I$ and $\boldsymbol{C}_B$;
2. then, a comparable cost is required for the application of the presented adjoint method as shown in table 3.3.

Table 3.3: Systems that demands the computation of a solution during the application of RBF-FD and the adjoint method explained in section 3.3.1. Those of dimension $m + M$ are solved once for each stencil in the physical domain

Method	Solved systems	System dimension
RBF-FD	$\boldsymbol{C}_I \boldsymbol{u}_I = f - \boldsymbol{C}_B \boldsymbol{u}_B$	N_I
	$\boldsymbol{M_{BC}}^T \boldsymbol{c}(\boldsymbol{x}_i) = \boldsymbol{h}$ for $i = 1, \ldots, N_I$	$m + M$
Adjoint	$\boldsymbol{C}_I^T \boldsymbol{\lambda}_1 = \frac{\partial J}{\partial \boldsymbol{u}_I}$	N_I
	$\boldsymbol{M_{BC}} \boldsymbol{\lambda}_{2,i} = \begin{bmatrix} \tilde{\boldsymbol{u}}_i \\ \boldsymbol{0} \end{bmatrix}$ for $i = 1, \ldots, N_I$	$m + M$

3.3.2 3D

Chapter 4

Results

4.1 Poisson Equation

Given a region $\Omega \subset \mathbb{R}^d$, with $d \in \mathbb{N}$ and bounded by the boundary $\partial\Omega$, the generic heat equation with internal heat generation [9], at steady state, is described by the Poisson Equation which is defined by the following Partial Differential Equation (PDE):

$$\begin{cases} -\Delta u(\boldsymbol{x}) = q(\boldsymbol{x}) & \text{in } \Omega \\ \mathcal{B}u(\boldsymbol{x}) = g(\boldsymbol{x}) & \text{on } \partial\Omega \end{cases} \tag{4.1.1}$$

where Δ indicates the Laplacian operator, $\mathcal{B}$ is the (possibly differential) linear operator that enforce the boundary conditions (BCs) and q and g are known functions. Clearly Poisson Equation is one of the many PDE that can be solved using meshless methods, including the RBF-FD discussed in this thesis.

We decided to evaluate the results of the adjoint method applied to RBF-FD method on the aforementioned equation due to both its wide range of applications in different areas (e.g. electrostatic, chemistry, gravitation and others) and the simplicity of its analytical manipulation.

In particular for this thesis and to obtain the results presented in this chapter we implemented RBF-FD method as follow:

- use RBFs augmented with polynomial terms as basis functions B_k to approximate the solution u in a similar way to what has been done in [10];
- formulate the problem in its strong form, so we will use the collocation technique in combination with the Weighted Residual Method.

thus we will apply the RBF-generated Finite Differences (RBF-FD) [11, 12] to solve (4.1.1) in one and three dimensional physics domain.

Mono dimensional physics domain are used to gain confidence with the techniques explained while three dimensional domain are used for their application in real case problems as those faced at Esteco. 2D cases, instead, have been skipped in the analysis since they would only be used to bridge the 2 previous scenarios and therefore would have had no practical utility.

RBFs are used for scattered data interpolation both for their physical foundation[overview 20, Hardy 1990] and for their profitable use in applications like meteorology, turbulence analysis and neural network [1]

Finally, collocation technique is employed because thanks to its ability to discretize the Boundary Value problem (4.1.1) expressed in strong form leads to a real fully meshless approach [13].

4.2 1D case

In this case $d = 1$, the physical domain is simply a segment and its boundary consists on its two endpoints that we indicate with a and b, both belonging to $\mathbb{R}$. Therefore we have $\Omega =]a, b[$ and $\partial\Omega = \{ a, b \}$. With respect to the rest of the boundary value problem we define $q(x) = -\omega^2 \sin(x)$ and we impose the following Dirichlet BCs: $u(a) = q(a)$ and $u(b) = q(b)$. Fixing the values $\omega = 1$, $a = 0$ and $b = \pi$ allows to rewrite the Poisson equation reported in (4.1.1) as

$$\begin{cases} -\Delta u(\boldsymbol{x}) = -\sin(x) & \text{in }]a, b[\\ u(a) & = 0 \\ u(b) & = 0 \end{cases} \tag{4.2.1}$$

In order to find an approximated solution to the problem (4.2.1) we employ the RBF-FD method parametrized as follow:

- $\varphi(r) = r^3$ Polyharmonic function is used as basic function to define the RBF interpolant;
- $P = 2$ is the degree of its polynomial augmentation;
- $N = 41$ evenly spaced nodes are used for domain discretization: $N_I = 39$ are placed in Ω and the other $N_B = 2$ are placed repsectivelly in a and b;
- $m = 7$ is the number of nodes which constitute a single stencil;

As done previously we indicate with $\boldsymbol{u}_I$ the vector obtained by solving the system (3.3.1) derived from RBF-FD discretization. The discretized version Equation (4.2.1) will be the constraints during the optimization

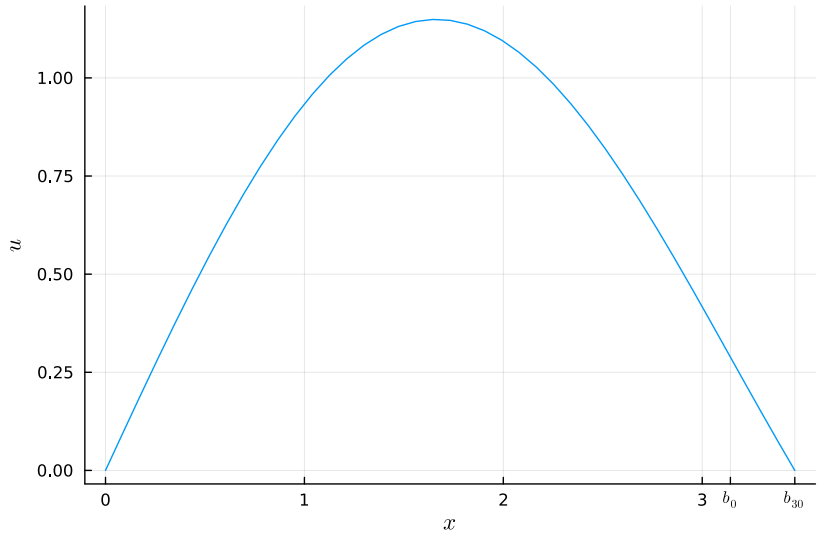


Figure 4.1: discretized approximated solution obtained solving (4.2.1) via RBF-FD at the end of the design optimization. b_0 and b_{30} indicate respectively the intial and the final position of the b endpoint

The design optimization problem, related to segment $[a, b]$, that we want to solve is the minimization of the following cost function:

$$J(\boldsymbol{u}_I, b) = \frac{1}{2}\frac{b}{N-1}\left(\sum_{i=2}^{40} u(x_i) - E\right)^2 \tag{4.2.2}$$

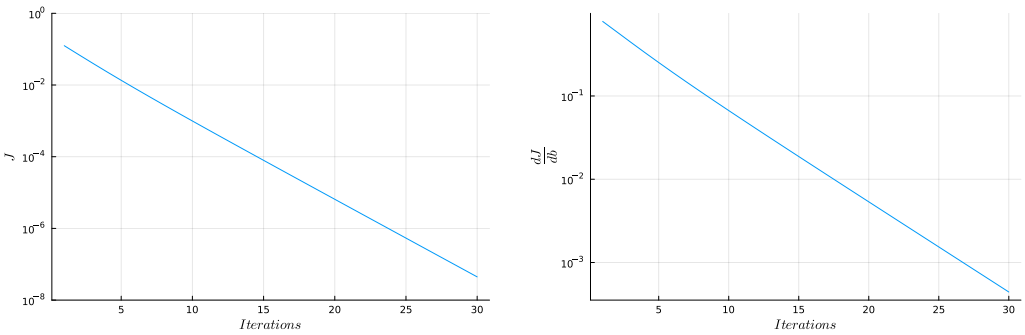
where $\boldsymbol{u}_I$ is the result of the where it can be seen the single design variable, $l = b$ and the result of th esingle which means that in order to meet the objective we are allowed to vary the length of the segment $[a, b]$. The discretized version Equation (4.2.1) will be the constraints during the optimization

The results after 30 steps in the optimization procedure are shown in figure 4.1.

while in figure 4.2 are shown the dynamic of the cost function and of its gradient respect the design variable b.

In this case we would like to modify the For the experiments we initially imposed $a = 0$ and $b = \pi$. In this case we have only one design parameter l that let modify the shape of the physical domain and it is the position of the endpoint b. The whole system can be thought as a bar whose length can be varied.

4.3 3D case



(a) *Cost function values* **(b)** *Cost function gradient values*

Figure 4.2: Trajectories of the cost function and its gradient throughout the optimization process

Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.4 Lorem Ipsum

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

4.4.1 Dolor sit amet

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Appendix A

Sit Amet

Bibliography

[1] Michael Hillman Jiun-Shyan Chen and Sheng-Wei Chi. "Meshfree Methods: Progress Made after 20 Years". In: *Journal of Engineering Mechanics* (2017).

[2] T. Belytschko et al. "Meshless methods: An overview and recent developments". In: *Computer Methods in Applied Mechanics and Engineering* (1996).

[3] W. Benz and E. Asphaug. "Simulations of brittle solids using smooth particle hydrodynamics". In: *Computer Physics Communications* (1995).

[4] S. Jun W. K. Liu and Y. F. Zhang. "Reproducing kernel particle methods". In: *International Journal for Numerical Methods in Fluids* (1996).

[5] P. Lancaster and K. Salkauskas. "Surfaces Generated by Moving Least Squares Methods". In: *Mathematics of Computation* (1981).

[6] E. J. Kansa. "Multiquadrics—A scattered data approximation scheme with applications to computational fluid-dynamics—I surface approximations and partial derivative estimates". In: *Computers & Mathematics with Applications* (1990).

[7] E. J. Kansa. "Multiquadrics—A scattered data approximation scheme with applications to computational fluid-dynamics—II solutions to parabolic, hyperbolic and elliptic partial differential equations". In: *Computers & Mathematics with Applications* (1990).

[8] M. A. Schweitzer. *Partition of Unity Method*. Lecture Notes in Computational Science and Engineering. 2003. URL: https://doi.org/10.1007/978-3-642-59325-3_2.

[9] H. Brezis. *Functional Analysis, Sobolev Spaces and Partial Differential Equations*. 2011. URL: https://doi.org/10.1007/978-0-387-70914-7.

[10] G. R Liu and Y. T. GU. *An Introduction to Meshfree Methods and Their Programming*. Cambridge University Press, 2005.

[11] B. Fornberg and N. Flyer. *A Primer on Radial Basis Functions with Applications to the Geosciences*. 2015.

[12] B. Fornberg and N. Flyer. "Solving PDEs with radial basis functions". In: *Acta Numerica* (2015).

[13] D. Miotti, R. Zamolo, and E. Nobile. "A Fully Meshless Approach to the Numerical Simulation of Heat Conduction Problems over Arbitrary 3D Geometries". In: *Energies* (2021).

[14] I. Amidror. "Scattered data interpolation methods for electronic imaging systems: a survey". In: *Journal of Electronic Imaging* (2002).

[15] J. C. Mairhuber. "On Haar's Theorem Concerning Chebychev Approximation Problems Having Unique Solutions". In: *Proceedings of the American Mathematical Society* (1956).

[16] R. Schaback and H. Wendland. "Kernel Techniques: From Machine Learning to Meshless Methods". In: *Acta Numerica* (2006).

[17] R. Zamolo. "Radial Basis Function-Finite Difference Meshless Methods for CFD Problems". PhD thesis. Università degli studi di Trieste, 2018.

[18] G. Fasshauer. *Meshfree Approximation Methods with Matlab*. World Scientific Publishing Company, 2007.

[19] D. Miotti. "Stable meshless methods for 3D CFD simulation on complex geometries based on Radial Basis Functions". PhD thesis. Università degli studi di Trieste, 2024.

[20] A. Tolstykh. *On using RBF-based differencing formulas for unstructured and mixed structured-unstructured grid calculations*. 2000. URL: https://scholar.google.com/scholar_lookup?title=On+using+RBF-based+differencing+formulas+for+unstructured+and+mixed+structured%E2%80%93unstructured+grid+calculations&author=Tolstykh+A.+I.&publication+year=2000&journal=Proc.+16th+IMACS+World+Congress&volume=228&pages=4606-4624.

[21] L. A. Bueno, E. A. Divo, and A. J. Kassab. "A coupled localized RBF meshless/DRBEM formulation for accurate modeling of incompressible fluid flows". In: *International Journal of Computational Methods and Experimental Measurements* (2017).

[22] L. A. Bueno, E. A. Divo, and A. J. Kassab. "Multi-scale cardiovascular flow analysis by an integrated meshless-lumped parameter model". In: *International Journal of Computational Methods and Experimental Measurements* (2018).

[23] G. Kosec and J. Slak. "Rbf-fd based dynamic thermal rating of overhead power lines". In: *Advances in Fluid Mechanics XII* (2018).

[24] G. Kosec and J. Slak. "Radial basis function-generated finite differences solution of natural convection problem in 3D". In: *AIP Conference Proceedings 2293* (2020).

[25] R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-dependent Problems*. Society for Industrial and Applied Mathematics, 2007.

[26] J. L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* (1975).

[27] A. Kolar-Požun et al. "Oscillatory behaviour of the RBF-FD approximation accuracy under increasing stencil size". In: *Computational Science – ICCS 2023* (2023).

[28] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul. "Automatic Differentiation in Machine Learning: a Survey". In: *Journal of Machine Learning Research* (2018).

[29] The MathWorks Inc. *What Is Design Optimization?* 2024. URL: https://it.mathworks.com/discovery/design-optimization.html.

[30] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Reading, Massachusetts: Addison-Wesley, 1993.

[31] R. Zamolo, D. Miotti, and E. Nobile. "Accurate stabilization techniques for RBF-FD meshless discretizations with neumann boundary conditions". In: *SSRN Electronic Journal* (2022).

[32] D. Rumelhart, G. Hinton, and R. Williams. "Learning representations by back-propagating errors". In: *Nature* (1986).

[33] L. Bacer. "Fully meshless approaches for the numerical solution of partial differential equations: radial basis function finite difference method and physics-informed neural network". MA thesis. Università degli studi di Trieste, 2023.

[34] K. C. Jeyanthi. "Efficient node generation over complex 3D geometries for meshless RBF-FD method". MA thesis. Università degli studi di Trieste, 2022.

[35] Wikipedia contributors. *Finite difference method*. Online; accessed 21-February-2024. 2024. URL: https://en.wikipedia.org/wiki/Finite_difference_method#Further_reading.

Ei fu. Siccome immobile,
Dato il mortal sospiro,
Stette la spoglia immemore
Orba di tanto spiro

Alessandro Manzoni – *Il Cinque Maggio*