Parallel Solver for the Poisson Equation on a Hierarchy of Superimposed Meshes, under a Python Framework

Federico Tesser

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR DE

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHEMATIQUES ET D'INFORMATIQUE DE BORDEAUX

Mathématiques Appliquées et Calcul Scientifique

Par Federico Tesser

Solveur Parallèle pour l'Equation de Poisson sur Mailles Superposées et Hiérarchiques, dans le Cadre du Langage Python

Sous la direction de : Angelo Iollo et Michel Bergmann

Soutenue le 11/09/2018

Président du jury : Professeur Galusinski Cedric

Membres du jury :

M. Berrone, Stefano	Professeur, Politecnico di Torino	Rapporteur
M. Galusinski, Cedric	Professeur, Université de Toulon	Rapporteur
M. Rougier, Nicolas P.	Chargé de recherche, Inria Bordeaux	Examinateur
M. Cisternino, Marco	Software Developer, PhD, Optimad Engineering	Examinateur
M. Bruneau, Charles-Henry	Professeur, Université de Bordeaux	Invité
M. Barthou, Denis	Professeur, Bordeaux INP	Invité

Titre : Solveur Parallèle pour l’Equation de Poisson sur Mailles Superposées et Hiérarchiques, dans le Cadre du Langage Python

Résumé : Les discrétisations adaptatives sont importantes dans les problèmes de flux compressible/incompressible puisqu'il est souvent nécessaire de résoudre des détails sur plusieurs niveaux, en permettant de modéliser de grandes régions d'espace en utilisant un nombre réduit de degrés de liberté (et en réduisant le temps de calcul). Il existe une grande variété de méthodes de discrétisation adaptative, mais les grilles cartésiennes sont les plus efficaces, grâce à leurs stencils numériques simples et précis et à leurs performances parallèles supérieures. Et telles performance et simplicité sont généralement obtenues en appliquant un schéma de différences finies pour la résolution des problèmes, mais cette approche de discrétisation ne présente pas, au contraire, un chemin facile d'adaptation. Dans un schéma de volumes finis, en revanche, nous pouvons incorporer différents types de maillages, plus appropriées aux raffinements adaptatifs, en augmentant la complexité sur les stencils et en obtenant une plus grande flexibilité. L'opérateur de Laplace est un élément essentiel des équations de Navier-Stokes, un modèle qui gouverne les écoulements de fluides, mais il se produit également dans des équations différentielles qui décrivent de nombreux autres phénomènes physiques, tels que les potentiels électriques et gravitationnels. Il s'agit donc d'un opérateur différentiel très important, et toutes les études qui ont été effectuées sur celui-ci, prouvent sa pertinence. Dans ce travail seront présentés des approches de différences finies et de volumes finis 2D pour résoudre l'opérateur laplacien, en appliquant des patchs de grilles superposées où un niveau plus fin est nécessaire, en laissant des maillages plus grossiers dans le reste du domaine de calcul. Ces grilles superposées auront des formes quadrilatérales génériques. Plus précisément, les sujets abordés seront les suivants: 1) introduction à la méthode des différences finies, méthode des volumes finis, partitionnement des domaines, approximation de la solution; 2) récapitulatif des différents types de maillages pour représenter de façon discrète la géométrie impliquée dans un problème, avec un focus sur la structure de données octree, présentant PABLO et PABLitO. Le premier est une bibliothèque externe utilisée pour gérer la création de chaque grille, l'équilibrage de charge et les communications internes, tandis que la seconde est l'API Python de cette bibliothèque, écrite ad hoc pour le projet en cours; 3) la présentation de l'algorithme utilisé pour communiquer les données entre les maillages (en ignorant chacune l'existence de l'autre) en utilisant les intercommunicateurs MPI et la clarification de l'approche monolithique appliquée à la construction finale de la matrice pour résoudre le système, en tenant compte des blocs diagonaux, de restriction et de prolongement; 4) la présentation de certains résultats; conclusions, références. Il est important de souligner que tout est fait sous Python comme framework de programmation, en utilisant Cython pour l'écriture de PABLitO, MPI4Py pour les communications entre grilles, PETSc4py pour les parties assemblage et résolution du système d'inconnues, NumPy pour les objets à mémoire continue. Le choix de ce langage de programmation a été fait car Python, facile à apprendre et à comprendre, est aujourd'hui un concurrent significatif pour l'informatique numérique et l'écosystème HPC, grâce à son style épuré, ses packages, ses compilateurs et pourquoi pas ses versions optimisées pour des architectures spécifiques.

Mots clés : Python, Différences Finies, Volumes Finis, Programmation Parallèle, Opérateur de Laplace, Discrétisations Adaptatives

---

Equipe de Recherche **MEMPHIS**

Title : Parallel Solver for the Poisson Equation on a Hierarchy of Superimposed Meshes, under a Python Framework

Abstract : Adaptive discretizations are important in compressible/incompressible flow problems since it is often necessary to resolve details on multiple levels, allowing large regions of space to be modeled using a reduced number of degrees of freedom (reducing the computational time). There are a wide variety of methods for adaptively discretizing space, but Cartesian grids have often outperformed them even at high resolutions due to their simple and accurate numerical stencils and their superior parallel performances. Such performance and simplicity are in general obtained applying a finite-difference scheme for the resolution of the problems involved, but this discretization approach does not present, by contrast, an easy adapting path. In a finite-volume scheme, instead, we can incorporate different types of grids, more suitable for adaptive refinements, increasing the complexity on the stencils and getting a greater flexibility. The Laplace operator is an essential building block of the Navier-Stokes equations, a model that governs fluid flows, but it occurs also in differential equations that describe many other physical phenomena, such as electric and gravitational potentials, and quantum mechanics. So, it is a very important differential operator, and all the studies carried out on it, prove its relevance. In this work will be presented 2D finite-difference and finite-volume approaches to solve the Laplacian operator, applying patches of overlapping grids where a more fined level is needed, leaving coarser meshes in the rest of the computational domain. These overlapping grids will have generic quadrilateral shapes. Specifically, the topics covered will be: 1) introduction to the finite difference method, finite volume method, domain partitioning, solution approximation; 2) overview of different types of meshes to represent in a discrete way the geometry involved in a problem, with a focus on the octree data structure, presenting PABLO and PABLitO. The first one is an external library used to manage each single grid's creation, load balancing and internal communications, while the second one is the Python API of that library written ad hoc for the current project; 3) presentation of the algorithm used to communicate data between meshes (being all of them unaware of each other's existence) using MPI inter-communicators and clarification of the monolithic approach applied building the final matrix for the system to solve, taking into account diagonal, restriction and prolongation blocks; 4) presentation of some results; conclusions, references. It is important to underline that everything is done under Python as programming framework, using Cython for the writing of PABLitO, MPI4Py for the communications between grids, PETSc4py for the assembling and resolution parts of the system of unknowns, NumPy for contiguous memory buffer objects. The choice of this programming language has been made because Python, easy to learn and understand, is today a significant contender for the numerical computing and HPC ecosystem, thanks to its clean style, its packages, its compilers and, why not, its specific architecture optimized versions.

Keywords : Python, Finite Differences, Finite Volumes, Parallel Programming, Laplace Operator, Adaptive Discretizations

---

Research Team **MEMPHIS**

Résumé substantiel : Dans la vie de tous les jours, nous traitons souvent des phénomènes multi-échelles, sans souvent le réaliser. Notre société elle-même est organisée selon une structure hiérarchique qui suit un chemin multi-échelle: pays, régions, villes, arrondissements et, à la fin, nous.

En laissant de côté nos aspects sociaux et en nous concentrant sur les phénomènes physiques, un outil d'analyse important consiste à les décomposer dans leurs différentes échelles.
Du point de vue de la physique en fait, tous les matériaux à l'échelle microscopique sont composé des noyaux et des électrons, dont la structure et la dynamique sont responsable du comportement macroscopique du matériau, tel que le transport, propagation d'onde, déformation.

L'avènement de l'informatique parallèle a contribué au développement de la modélisation multi-échelle, pas seulement en théorie. Puisque plus de degrés de liberté pourraient être résolus par des environnements informatiques parallèles, des formulations algorithmiques plus précises pourraient être admises.
Cependant, pour étudier certains phénomènes, des schémas d'ordre élevé et plus précis ne suffisent pas, parfois. Donc, une approche alternative/coopérant est de réduire la dimension caractéristique du domaine de calcul, pour essayer de reproduire tous les composants à leurs différentes échelles. Et ce faisant, implique de plus grandes moles de données, qui nécessitent plus de puissance de calcul et plus de temps à traiter.
Mais le pouvoir de calcul n'est pas illimité (comme il est pas notre temps) et, surtout, il ne vient pas gratuitement. Les discrétisations adaptatives sont importantes dans de nombreux problèmes multi-échelle, où il est essentiel de réduire la faim computationnelle et le temps de calcul tout en obtenant la même ou plus grande précision, dans des régions particulières du domaine de calcul.

En mathématiques appliquées, beaucoup de méthodes sont bien adaptées pour être utilisées sur des grilles régulières (et en particulier cartésiennes), car elles sont simples et permettent une représentation claire du domaine considéré.
Le principal inconvénient de cette approche est qu'elles ne permettent pas de discrétisations adaptives, en forçant l'utilisateur à affiner le domaine de calcul globalement, sans suivre la forme, par exemple, des frontières physiques et des corps à l'intérieur du contexte analysé.
Par conséquent, les métriques nécessaires pour passer de ces systèmes de référence bodyfitted à ceux cartésiens, sont un peu ennuyeux à gérer, ainsi que nécessaire.

Pour cette raison l’AMR (raffinement de maillage adaptatif) est vital, surtout pour les grands calculs: supposons de vouloir «donner un sens à l'univers avec des supercalculateurs», comme a dit Tom Abel de l'Université de Standford: c'est un problème cosmologique qui incorpore la «mère de toutes les échelles», avec des variations spatiales de l'ordre de 10^12 plage dynamique jusqu'à 10^15.
Et en essayant de le résoudre en utilisant une grille raffinée uniforme, il faudrait une vie, ainsi que d'énormes ressources de calcul. Et puisque les solutions complexes nécessitent plus de mémoire, des outils adaptatifs efficaces comme l’AMR sont essentiels (bien que l’AMR ne soit qu'un outil parmi d'autres qui doivent être utilisés).

Après une brève introduction sur les techniques AMR, et en passant dans un section consacrée aux Chimera grids, le premier chapitre examinera le SAMR (raffinement de maillage adaptatif structuré) et certains logiciels connexes, et se terminera avec l'introduction d'une nouvelle méthode adaptative, sujet de ce manuscrit.

En particulier, ce travail présentera une nouvelle méthode adaptative de différences finies et de volumes finis 2D pour discrétiser l'opérateur laplacien sur un domaine de calcul constitué de plusieurs quadrilatères génériques qui se chevauchent.
Cette approche peut être insérée dans la typologie chimère, étant chaque grille construite seule, sans l'intrusion des autres, et nécessitant une couche de communication pour la résolution finale. Contrairement à ce qui se passe dans les approches SAMR (où les communications sont nécessaires à chaque niveau de «l'arbre télescopique» des taches construites sur la plus grossière), ici une seule communication est nécessaire, avec un seul niveau de chevauchement, exploitant l'affinage intrinsèque de la structure de données octree.
Caractéristique de cette approche est d'‘etre pensé et implémenté en Python, comme langage de programmation.

Les principaux objectifs de ce travail peuvent donc être résumés comme suit:

- appliquez une nouvelle approche pour les problèmes multi-échelles;
- résoudre les difficultés présentes dans les méthodologies cartésiennes, chimères et SAMR, en laissant les avantages inchangés, en utilisant des maillages octree: pas de changements dans les équations descriptives du phénomène considéré; affiner le maillage là où c'est nécessaire; simplifier les algorithmes de communication entre les patches;
- écrire un code HPC entièrement parallèle;
- créer une bibliothèque Python accessible gratuitement;
- augmenter le nombre d'enveloppes scientifiques écrites pour Python et déjà présent (MPI4Py, NumPy, PETSc4py, etc.), avec un framework numérique complet, Python compatible;
- explorer l'écosystème HPC en utilisant de nouvelles techniques et langages de programmation.

Et le thème principal du chapitre deux c’est précisément une introduction à Python, un langage de programmation de haut niveau et interprété, destiné à la programmation general-purpose, qui fournit des concepts qui permettent un codage clair à petites et grandes échelles, grâce à son système de type dynamique et sa gestion automatique de la mémoire.

Créé par Guido van Rossum et relaché en 1991, Python a une philosophie de élaboration qui met l'accent sur la visibilité du code, et une syntaxe qui permet aux programmeurs d'exprimer des concepts dans moins de lignes de code qu'avec d'autres langues. Il prend en charge plusieurs paradigmes de programmation, y compris objet oriented, fonctionnel et procédurale, et dispose d'une bibliothèque standard vaste et complète.
De plus, ses interpréteurs sont disponibles pour de nombreux systèmes d'exploitation.

Mais, étant qu'il s'agit d'un langage interprété, il est souvent beaucoup plus lent que les langages compilés. En effet, les performances restent plus lentes par rapport aux langages plus anciens tels que C et C ++.
À cet régard, la dernière partie du chapitre montre un outil qui vient à l'aide de Python, Cython, un surensemble performant de Python qui peut être l'amalgame entre son sous-ensemble et l'écosystème HPC, pour surmonter cette écart entre les langages de programmation compilés et interprétés. Certains résultats numériques montrent des performances en utilisant du code Python pur, un code Cython mixte ou une implémentation pure en C++.

Le troisième chapitre, par contre, il présente une vue d'ensemble des structures de données hiérarchiques pour représenter des images, telles que le quadtree et l'octree (les octrees sont l'analogue 3D des quadtrees). Ces deux types de données fondamentaux sont basés sur le principe de la décomposition récursive et leurs principaux champs d'utilisation sont: la représentation des données utilisées dans les applications en infographie, robotique et computer vision.

Un quadtree est une structure de données arborescente dans laquelle chaque nœud interne a exactement quatre enfants. Les carrés sont le plus souvent utilisés pour partitionner un espace 2D en le subdivisant de manière récursive en quatre quadrants ou régions. Les données associées à une feuille varient en fonction de l'application, mais la feuille représente une « unité des information spatiale intéressant ».
Les régions subdivisées peuvent être carrées ou rectangulaires, et la résolution de la décomposition (c'est-à-dire, le nombre de fois que le processus de décomposition est appliqué) peut être fixé au préalable ou peut être régie par les propriétés des données d'entrée.

Sur la base de ce qui précède, les quadtrees peuvent être différenciés sur les bases suivantes:

- le type de données qu'ils représentent;
- le principe guidant le processus de décomposition;
- la résolution (variable ou non);
- si la forme de l'arbre est indépendante de l'ordre dans lequel les données sont traitées.

Le chapitre explique la construction de quadtrees, et leurs types les plus courants.

Bien sûr, étant un manuscrit sur les mathématiques appliquées, beaucoup d'importance est consacrée à l'utilisation des quadtrees (et de leur contrepartie 3D) dans la décomposition de l'espace. En fait, l'une des motivations pour le développement de structures de données hiérarchiques, est un désir de économiser des espace. Outre la prise en compte des critères de feuille, l'étude des structures de données hiérarchiques a également porté sur la manière de coder l'arbre représentant la hiérarchie.
Et dans ce contexte, l'approche sur lequel le travail se concentre est celui basé sur les locational codes, d'abord proposé par Morton, et connu en fait comme «Morton indices».

En analyse mathématique et informatique, cette représentation binaire est aussi appelée ordre Z, ordre Morton, ou code Morton, et est un fonction qui mappe des données multidimensionnelles dans une dimension, tout en préservant la localité des points de données. L'ordre Z peut être utilisé pour construire efficacement un quadtree pour un ensemble de points. L'idée de base est de trier l'ensemble d'entrée selon l'ordre Z et, une fois triés, les points peuvent être stockés dand un arbre binary search et utilisé directement (quadtree linéaire), ou ils peuvent être utilisés pour construire un quadtree pointer based.
Comme alternative, la courbe de Hilbert a été suggérée, ayant un meilleur comportement de conservation de l'ordre, et donnant ainsi une meilleure localisation spatiale. Mais les calculs pour la distance de Hilbert sont plus compliqués que pour le calcul de Morton, en conduisant à important surcharge pour le processeur.
Comportement qui, dans les applications liées au CPU, n'est pas très apprécié.

le chapitre se termine par un paragraphe dédié à l'utilisation de les octrees pour la génération du maillage, et à l'introduction de la bibliothèque Python créée par l'auteur pour le projet en question : PABLitO.

Bibliothèque qui est décrite plus précisément dans le chapitre suivant: ceci et le dernier sont les chapitres où il est couvert en détail la nouvelle méthode adaptative qui est le question clé de cette thèse.
Méthode qui vise à réduire 1) la complexité typique des approches Chimera (où l'algorithme global ne peut pas être mis en œuvre si facilement), et 2) le nombre croissant de grilles de calcul en place (qui peuvent être un point chaud de complexité pour le schéma de communication entre toutes les mailles). Et bien sûr, cette méthode vise à marquer un point d’inflexion dans les techniques de programmation pour la CFD, en laissant ouverte une lueur sur l'utilisation de langages de programmation non classiques (bien sûr, relativement parlant), ayant été complètement développé en utilisant Python et Cython.
Ces deux langues ont été utilisées dans le but de présenter un véritable application informatique totalement opérationnelle, comme déjà dit, avec un langage (Python) adapté pour un développement et formations faciles.

Après avoir présenté les principaux points de modélisation et de simulation de processus physiques, et après avoir établi les points clés pour la simulation numérique des équations aux dérivées partielles (EDP) et pour le méthodes numériques de différences et de volumes finis, ces chapitres abordent les aspects techniques et les aspects détaillés de cette méthode de discrétisation 2D basée sur le chevauchement des quadrilatères génériques faites par quadtrees qui, comme précédemment mentionné, peut être comparé à l’AMR structuré à bloques, mais que diffère en réalité dans l'usage de la capacité intrinsèque de les quadtrees eux-mêmes de pouvoir être raffinés localement, en utilisant par exemple un estimateur d'erreur ou des fonctions de prédiction, ou peut-être juste des fonctions de forme.

Approches algorithmiques et de programmation pour les communications parallèles entre les processus sont présentés, ainsi que l'idée derrière la construction du système monolithique qui sera résolu. Des exemples sont montrés, pour illustrer également les différences et l'utilisation des inter et intra communicateurs MPI.

L'équation de Poisson a été introduite, ainsi que les mécanismes Python pour appliquer la sérialisation sur les données définies par l'utilisateur, nécessaires pour communiquer entre de multiples processus et pour résoudre, en parallèle, des problèmes de calcul intensif avec le paradigme MPI (malgré qu'il ne soit pas le seul disponible dans l'offre Python pour la concurrence).
En outre, un exemple pratique du travail de programmation effectué à niveau Python a été montré, en utilisant les extensions Cython écrites ad hoc, avec les approches algorithmiques et de programmation pour le chevauchement des générique quadrilatère patches, utilisés pour résoudre l’'équation de Poisson 2D, a été décrite.

Le dernier chapitre applique la théorie du chapitre précédent, en montrant des résultats numériques sur différentes mailles, allant de une superposition classique à la déformation et au raffinement intrinsèque.
Les paragraphes concernant ce sujet est toutefois reporté aux sections relatif à:

- l'explication mathématique de certaines interpolations utilisées dans les exemples;
- l'explication de la théorie dans la littérature qui concerne les résultats théoriques gouvernant l'ordre des interpolations sur les régions frontalières des différents patches de quadtree.

Le chapitre sera suivi par les conclusions couvrant tout le travail effectué, et qui vont prendre les fils des sujets couverts, et des questions non encore terminé.

This thesis aims to be a common ground between applied mathematics and information technology.
What will be read at its inside want to be clear and accurate for people from both the audiences. That's why technicalities, specific to these two "distinct" worlds, won't be around (not completely, at least).

A "vulgarisation" approach will be followed, explaining the basis of the subjects that will be addressed, going into depth sometimes, but remaining however in a common context for the previous two disciplines.
Context that is, for me of course, that thin line marking, although invisibly, the excessively large cut-off between numerical methods and software developing, which at first sight may seem two fields quite different, but that in reality could present several commonalities, more than the current ones.

Python represents one of these: a language widely used in software developing, thanks to its usage versatility but that, still, in the simulations field is not considered like a valid alternative to the bare-bones languages that dominate this scenario.
A real implementing case, so, could represent a first step in terms of acceptance and usage of a new language, in a new (for it) application context.

On the contrary, so many times can be seen IT frameworks, tools and instruments very powerful with "easy" mathematical implementations and concerns.
That's why it could be interesting to introduce in this field different approaches, to create here too valid alternatives and use cases to the mainstream patterns.
In this context, a parallel multi-block approach for the resolution of the Laplace operator could be a great starting point, gathering innovation with a long-established mathematical case study (and guess what you will find in pursuing the lecture).

Hoping that anyone looking through this thesis can find it clear and of pleasant reading and, why not, source of inspiration, I would like to thank the ones which have had the courage (willingly and not) to get, anyway, up to this point: my parents, my fiancée, my family, who have always been there, and have always put up with me. My supervisors, who have always believed in me, helped and cheered me on during this "journey". The commission, which have agreed on judging my job. My friends, who have not forgotten me during these years. To all of you, thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

AMR	Adaptive Mesh Refinement
SAMR	Structured Adaptive Mesh Refinement
PDE	Partial Differential Equation
CHOMBO	Software for Adaptive Solutions of Partial Differential Equations
SAMRAI	Structured Adaptive Mesh Refinement Application Infrastructure
FDM	Finite-Difference Method
FVM	Finite-Volume Method
HPC	High Performance Computing

In everyday life we often deal with multiscale phenomena, without often realizing it. Our society itself is organized under a hierarchical structure which follow a multiscale path: countries, regions, cities, districts and, at the end, us.
Leaving our social aspects and focusing on physical phenomena, an important analysis tool is to decompose them into their different scales.
From the view point of physics in fact, all materials at the microscale are made up of the nuclei and the electrons, whose structure and dynamics are responsible for the macroscale behavior of the material, such as transport, wave propagation, deformation.

The advent of parallel computing contributed to the development of multiscale modeling, not only in theory. Since more degrees of freedom could be resolved by parallel computing environments, more accurate and precise algorithmic formulations could be admitted.
However, to study certain phenomena, high order and more accurate schemes are sometimes not enough. So, an alternative/co-operating approach is to reduce the characteristic dimension of the computational domain, to try to reproduce all of the components at their different scales. And doing so, involves greater moles of data, which require more computational power and more time to be processed.

But computational power is not unlimited (as it is not our time) and, above all, it does not come free of charge. Adaptive discretizations are important in many multiscale problems, where it is critical to reduce the computational hunger and time while achieving the same or greater accuracy, in particular regions of the computational domain.

In applied mathematics, a lot of methods are well suited to be used on regular[1] (and in particular, Cartesian[2]) grids, because they are simple and allow a clear representation of the domain considered.

The main drawback of this approach is that it does not allow adaptive discretizations, forcing the user to refine the computational domain globally (see Figures 1.1a and 1.1b), without following the shape, for example, of the physical boundaries and bodies inside the analysed context.
Therefore, the metrics needed to switch from these body-fitted reference systems to the Cartesian one, are a bit annoying to handle, as well as necessary.

That's why AMR (adaptive mesh refinement) is vital, especially for big computations: suppose to want "to make sense of the universe with supercomputers", as Tom Abel of Standford University said: it is a cosmological problem that incorporates the "mother of all scales", with spatial variations on the order of 10^{12} and a dynamic range up to 10^{15}.
Thinking to try to solve it using a uniform refined grid, it would require a lifetime, as well as huge calculus resources. And since complex solutions

[1]A regular grid is a tessellation of n-dimensional Euclidean space by congruent parallelotopes (e.g. bricks).

[2]A Cartesian grid is a special case of a regular grid where the elements are unit squares or unit cubes, and the vertices are integer points.

require more memory, efficient adaptive tools like AMR are essential (although AMR is but one tool among many that are needed to be employed).

Following a brief introductory survey of AMR techniques, walking past a section dedicated to Chimera grids, the chapter will examine the SAMR (structured adaptive mesh refinement) techniques and some related software packages, and it will end with the introduction of a new adaptive method, that will be the topic of Chapter 4.

1.1 AMR Techniques

In numerical analysis, adaptive mesh refinement (or AMR, as we have already mentioned) is a method for adapting the accuracy of a solution within certain regions of simulation, in a dynamic (or static) way during the time the solution is being calculated.

When solutions are calculated numerically, they are often limited to predetermined quantified grids as in the Cartesian plane, which constitute the computational grid (or "mesh"). But many problems in numerical analysis do not require a uniform precision in the numerical grids used for the simulation, favouring an accuracy that could be refined only in the regions requiring the added details.

Adaptive mesh refinement procedure provides such a dynamic programming environment, to adapt the precision of the numerical computation based on the requirements of the relevant computation problem, and based on its specific areas which need more accurate precision, leaving the other regions at lower levels of accuracy, details and resolution.
This dynamic technique of adapting computational precision to specific requirements, has been accredited to Marsha Berger, Joseph Oliger, and Phillip Colella[3], who developed an algorithm for dynamic gridding called *local adaptive mesh refinement*. In a parallel computing context, an important consequence of the adaptation is that the dynamically changing resolution leads to a dynamically changing work load, data volume, and communication pattern at run-time.
This is called **dynamic load balancing**[4], and has implications for data

[3]Berger, M. J.; Colella, P. (1989). *Local adaptive mesh refinement for shock hydrodynamics.* J. Comput. Phys. (Elsevier) 82: 64–84.

[4]In computing, load balancing improves the distribution of workloads across multiple computing resources. It aims to optimize resource use, maximizing throughput, minimiz-

placement as well as parallelization granularity.

The local adaptive mesh refinement algorithm starts covering the entire computational domain with a coarse regular Cartesian grid. As the calculation progresses, individual grid cells are tagged for refinement, using a criterion that can either be user supplied, or automatic.
All tagged cells are then refined, which means that a finer grid is superimposed on the coarse one, and a correction procedure is implemented to correct the transfer along coarse-fine grid interfaces, to ensure that the amount of any conserved quantity, leaving one cell, exactly balances the amount entering the bordering cell.

The use of AMR has, since then, been used on a broad range of problems, especially regarding the study of turbulence and of large scale structures simulations (like, for example, in an astrophysics context), and has allowed scientists to solve problems that would have been completely intractable on a uniform grid.
And since then, different approaches and different techniques have been developed, having however the common denominator to reduce the computational efforts of the numerical simulations. In figure 1.1 are given the most common.

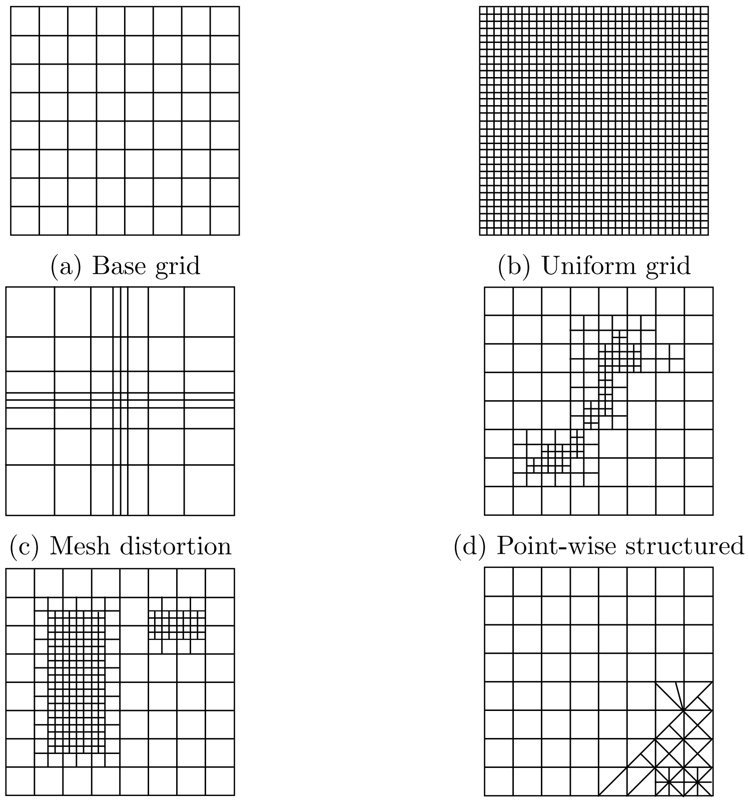
1.2 Chimera grids

Overset composite grids method, also known as the Chimera overset grids technique (named like this after the composite monster of Greek mythology[5]), have long been recognized as an attractive approach for treating problems with complex geometries.
However, there are cases where, generating body-fitted grids, remains labor intensive and error prone, and in this cases other approaches can be applied, as we will see in the next sections.

The solution process uses a grid system that discretizes the problem domain by using separately generated but overlapping grids, that need to update and exchange boundary information through proper interpolations.
This method has been used successfully over the last two decades[13] [17],

ing response time and avoiding the overloading of any single resource.

[5]The Chimera was, according to Greek mythology, a monstrous fire-breathing hybrid creature, composed of the parts of more than one animal.



(a) Base grid (b) Uniform grid

(c) Mesh distortion (d) Point-wise structured

(e) Block structured (f) Ustructured

Figure 1.1: AMR Refining methods representing different ways to refine meshes for an adaptive approach. Base grid is showed in the upper left corner, while in the subsequent subfigures is modified to follow the refining paths, expressed by the corresponding subcaptions

primarily to solve problems involving fluid flow in complex and dynamically moving geometries, allowing the introduction of very complex representations that, in a classic Cartesian grid, would be analysed via a Level Set or Immersed Boundary approach, imposing so some kinematic boundary conditions, relaxing the difficulty in the meshes generation, but not taking into account the reduction of the total number of cells making up the overall mesh.

The overset composite grids method[11] is a way of assembling multiple grids and treating them as a single grid.

Basically, this method consists in generating a set of structured[6] or unstructured[7] component grids that cover the computational domain and overlap where they meet, thus generating some partially overlapping blocks. The geometry of the components can be defined individually, and hence the grid around them can be generated separately.
In a full Chimera grid system, a complex geometry is therefore decomposed into a system of simpler overlapping grids, each of one covering less complex geometries.
Boundary information is then passed between these grids, using different interpolations of the flow variables; note that many gridpoints may not be used in the solution (*hole points*), while each block's boundary (or *fringe*) points, which lie in the interior of neighboring blocks, will require information from that containing block.

To summarise, in very general terms, there are three steps to setting up an overset simulation, as shown below.

1. **Grid generation**. The grids may be structured, unstructured, Cartesian, or a combination of these. One intuitive combination occurs when structured-curvilinear grids and Cartesian grids are used: body-fitting curvilinear grids are built independently for each geometric component, and then embedded within a coarser Cartesian grid (see Figure 1.2).
   Because each curvilinear grid is paired with a component from the geometry, overset grids can be used to track relative motion with computational efficiency, but domain connectivity must be performed dinamically, so that adjacent grids share information.

2. **Hole cutting**. In this step, grid points are eliminated in both the main grid and the component grids (as previously said, in a Chimera approach many gridpoints will not be used).
   First, all points in the component grids which are outside the computational domain, and all points which overlap with other component grids and which are not needed for the solution interpolation, are removed.

[6]Structured means that mesh points can be indexed in such a way that neighbor relations between points can be inferred from the indices

[7]An unstructured grid is identified by irregular connectivity. It cannot easily be expressed as a two-dimensional or three-dimensional array in computer memory.

Next, all points in the main grid that are overlaid by component grids and which are not needed for the interpolation are eliminated. Doing so, holes are created in the main grid.

3. **Chimera interpolation**. The main difficulty in the use of overset composite grids methods is in the data transfer between overlapping grids.
   Supposing a cell-centered finite volume scheme, we use the concepts of *donor* and *receptor* cells: considering two grids, Grid A and Grid B, that overlap with each other, a receptor point, say on Grid B, is a fringe point which needs to receive flow information from Grid A, to provide the boundary condition for its grid. The donor cell, instead (and for this particular receptor cell), is identified as the cell on Grid A that contains the receptor point (see Figure 1.3).
   A simple interpolation method consists in directly transfering the flow variables from the donor cell to the receptor cell, but with a more accurate interpolation of course, better result in the convergence order can be obtained.

To conclude this section, we report a minor recap of the pros and cons of the Chimera grids method. Approach that, of course, has a lot of advantages, like:

- capability to handle complex geometries;
- capability to reduce the time and the efforts to generate a grid;
- capability to allow, without further difficulties, evaluation of flows around moving bodies.

But it has also some disadvantages, as:

- more complications because of programming complexities and coupling of the grids;
- difficulties to mantain conservation at the interface;
- interpolation process may introduce errors of convergence problems, if the solution exhibits strong variation near the interface;
- involving frequent searches over the cells in different zones or component grids means that these searches could be very expensive for large grids.

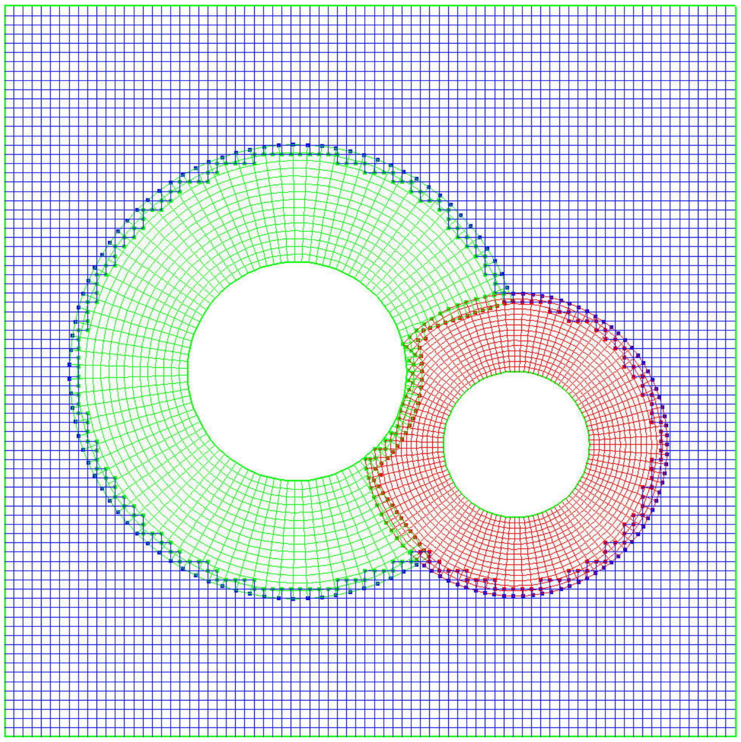


Figure 1.2: A simple overset composite grid where the hole points are blanked. Here are easily displayed the first two steps of the recap on setting up an overset simulation: *grid generation* (here curvilinear and Cartesian grids are used) and *hole cutting* (interior points of the components grids are eliminated). Image by Joel Guerrero[5]

1.3 Structured AMR

In the previous section we have introduced the Chimera approach, and we have mentioned some of its positive and negative aspects.
One of the most striking cons is that, operating with this approach, we are bound to the use of grids set at priori, thus making more difficult (if not impossible) to refine our computational domain following a time step.
Additionally, dealing with structured and unstructured grids at the same time, can introduce additional complications, like:

- Cache-reuse/vectorization nearly impossible (unstructured);
- necessity to store neighborhoods (unstructured);

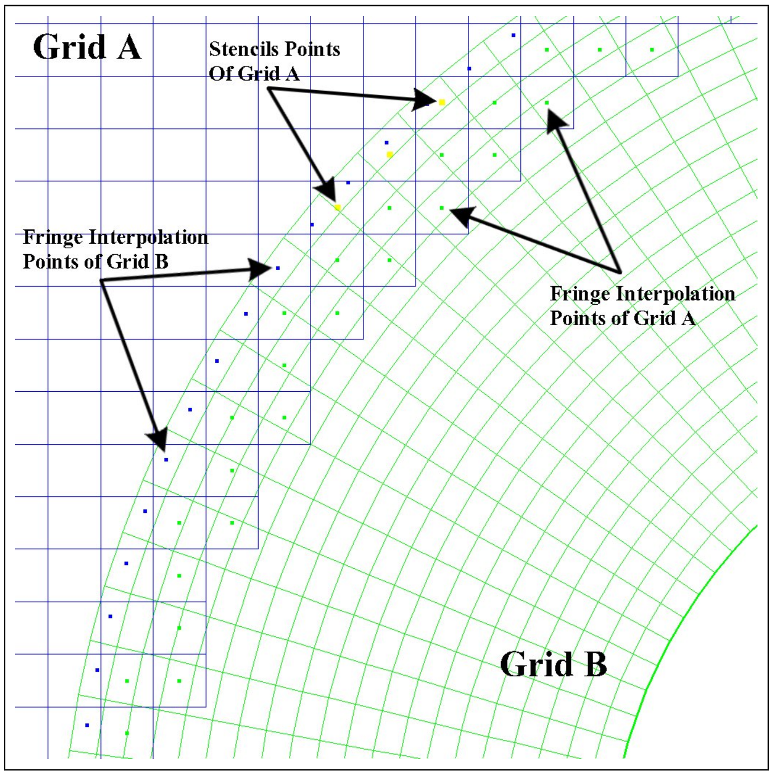

Grid A
Stencils Points Of Grid A
Fringe Interpolation Points of Grid B
Fringe Interpolation Points of Grid A
Grid B

Figure 1.3: Interpolation elements for overlapping grids. Image by Joel Guerrero[5], showing the third step of the overset simulation recap, where donor and receptor cells are set out

- complex load-balancing (unstructured);
- parent/child relations (structured);
- hanging nodes (structured).

Weighing up these issues, one can realise that an adaptive approach, totally structured, could not hurt.

In fact, what is particular about structured adaptive mesh refinement is that, normally, a structured mesh takes the form of a logically rectangular grid, where a numerical PDE solver can be implemented, using array data structures to represent the mesh points and their associated solution values. This is a very delicate aspect to solve PDE, because the key to efficiency

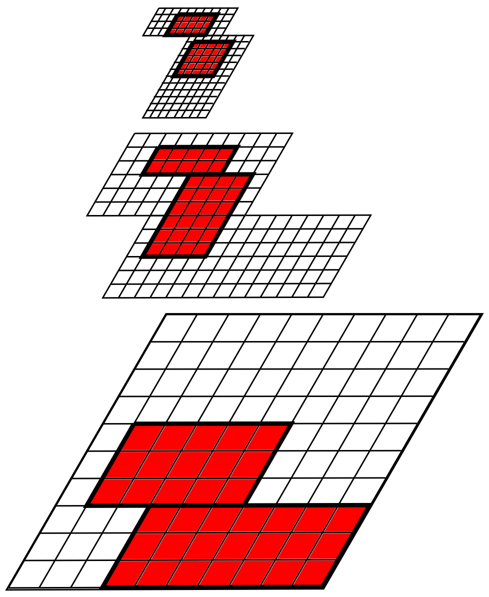


Figure 1.4: Telescopic representation of the block-structured adaptive mesh refinement. In red are highlighted the parts of each telescopy's level, partially refined in the next step, just on the surface concerned by the refining. The refined portions can be chosen using, for example, the Berger–Colella algorithm

in PDE solvers is that the neighbor relations between mesh points can be inferred from the array indices in the data structure used to store the mesh. Due to this property, the operation to retrieve the solution values at neighboring mesh points can be particularly efficiently implemented for structured meshes.

The most common approach to structured AMR is to adapt the computing coarse grid by adding refined rectangular grid patches in areas where higher resolution is required, removing them where this fine grain level is no longer necessary.
Essentially, this consists of four steps (Berger–Colella algorithm):

1. point-wise estimation of the errors in the computed solution;

2. flagging of points where the accuracy is insufficient;
3. Clustering of flagged points;
4. insertion of higher resolution grid patches around such clusters.

Iteratively repeating this procedure can provide a hierarchical, composite, structured grid, where each patch is inserted on the top of the underlying coarser grid. And of course, the four steps listed above contribute to the overall execution time of the simulation, and from their execution depends the trade-off between the gain in execution time (having reduced the number of grid points) and the payload due to the mesh adaptation.
In order to improve the payoff, an approach *block-wise* to SAMR has ben introduced in literature[2], whose purpose is to avoid clustering and grid-fitting steps, dividing the initial coarse grid into blocks and subsequently, if some points in a block are flagged for refinement, refining the whole block of belonging (see Figure 1.4).

SAMR solution methods, however, share characteristics with uniform, non-adaptive structured mesh methods. In particular, the simulation code may be organized as a collection of numerical routines that operate on data defined over logically-rectangular regions and communication operations that pass information between those regions, to fill "ghost cells".
Since a SAMR solution is constructed on a composite mesh, the numerical algorithms and approximations must treat internal mesh boundaries between coarse and fine levels properly to maintain a consistent solution state.

A large number of frameworks for solving partial differential equations using the technique of structured adaptive mesh refinement have been developed, and many of these are also freely available for downloading on the Internet. The vast part of these are implementing the Berger–Colella algorithm with a hierarchy of refinement levels on top of each other; what makes them different is the parallelization model.
The parallel implementation of SAMR application can indeed be realized in a distributed memory environment using explicit message passing, or in a global shared memory one, using a thred model. And the two implementations can also be combined into an hybrid approach, using message passing and threads together.
For the continuation of the manuscript, and for consistency with the work that will be presented, we mention just two frameworks, which belong to the first category of parallel implementation.

CHOMBO[8] provides a set of tools for implementing finite difference and finite volume methods for the solution of partial differential equations on block-structured adaptively refined rectangular grids.
Both elliptic and time-dependent modules are included, and it supports calculations in complex geometries with both embedded boundaries and mapped grids.

SAMRAI[9] (Structured Adaptive Mesh Refinement Application Infrastructure) is the code base in CASC (Center for Applied Scientific Computing) for exploring application, numerical, parallel computing, and software issues associated with SAMR.

1.4 Overlapping generic grids

We have already seen that in many problems in partial differential equations we are confronted with multiple length scales and strong spatial localizations.

Finite-difference[10] calculation using block-structured adaptive mesh refinement is a powerful tool for computing solutions to partial differential equations involving such multiple scales. In this approach, the underlying problem domain is discretized using a rectangular grid and a solution is computed on that grid.
Regions requiring additional resolution are identified by computing some local measure of the original error and covered by a disjoint union of rectangles in the domain, which are then refined by some integer factor. The solution is then computed on the composite grid, and this process may be applied recursively.

But, if this approach has multiple advantages (putting more grids where the solution is more interesting leaving elsewhere the grid coarse, saving order of magnitude in memory and in run time, etc.), it has also some drawbacks like, in particular, the algorithm and communication patterns complexities between levels.

[8]https://commons.lbl.gov/display/chombo/

[9]https://computation.llnl.gov/projects/samrai

[10]In mathematics, finite-difference methods (FDM) are numerical methods for solving differential equations by approximating them with difference equations, in which finite differences approximate the derivatives.

Moreover, when the grid is not cartesian, the discretization of the differential operators in space must take into account the metrics, making grid transformations a bit annoying to handle. This problem, imposed by physical domain body-fitted grids, can be crossed more easily using a finite-volume[11] approach on octree meshes.

This work will presents a new 2D adaptive finite-difference and finite-volume method to discretize the Laplacian operator on a computational domain made of multiple, overlapping, generic quadrilateral grids.
Looking at what previously said, this approach can be "fitted into" the Chimera typology, being each grid built alone, without the intrusion of the others, and needings a communication layer for the final resolution.
Unlike what happens in SAMR approaches, where communications are needed in every level of the "telescopic tree" of patches built on the coarser one, here only one communication is required, having just one level of overlapping, exploiting the intrinsic refining of the octree data structure.

The main objectives of this approach can be summarised as follows:

- Apply a new approach for multiscale problems.

- Solve those difficulties present in Cartesian, Chimera and SAMR methodologies, leaving the benefits unchanged, using octree meshes: no changes in descriptive equations of the phenomenon under consideration; refining the mesh where it is needed; simplify the communication alghorithms between patches.

- Write a fully-parallel *HPC* code.

- Create a free accessible Python library.

- Increase the number of scientific wrapper written for Python and already present (*MPI4Py*, *NumPy*, *PETSc4py*, and so on), with a full, Python compatible numerical framework.

- Explore the *HPC* ecosystem using new programming techniques and programming languages.

[11]Finite volume refers to the small volume surrounding each node point on a mesh. In the finite volume method, volume integrals in a partial differential equation that contain a divergence term are converted to surface integrals, using the divergence theorem. These terms are then evaluated as fluxes at the surfaces of each finite volume.

1.5 Summary

This chapter has introduced the classic resolution scheme of partial differential equations (PDE) on a uniform grid, and has shown that, for well behaved problems, this approach gives satisfactory results.
There are classes of problems, however, where, due to discontinuities, steep gradients, shocks, etc., the uniform spacing approach is computationally extremely costly, requiring different and more "ad-hoc" solutions. Besides, for time dependent problems, it is difficult to predict in advance a uniform mesh spacing that will give acceptable results.
That's why were presented different AMR techniques.

Bibliography

[1] K. Tomko Q. Liu A. Hamed, D. Basu. Performance characterization and scalability analysis of a chimera based parallel navier-stokes solver on commodity clusters. *Parallel Computational Fluid Dynamics 2005*, 2005.

[2] Ann S. Almgren. Introduction to block-structured adaptive mesh refinement (amr). Center for Computational Sciences and Engineering Lawrence Berkeley National Laboratory.

[3] R. W. Anderson B. T. N. Gunney. Advances in patch-based adaptive mesh refinement scalability. *Journal of Parallel and Distributed Computing*, 2015.

[4] Jaideep Ray Benjamin A. Allan, S. Lefantzi. The scalability impact of a component-based software engineering framework on a growing samr toolkit: a case study. *Parallel Computational Fluid Dynamics 2005*, 2005.

[5] Joel Guerrero. Overset composite grids for the simulation of complex moving geometries.

[6] A. M. Wissink D. A. Hysom Gunney, B. T. N. Parallel clustering algorithms for structured amr. *Journal of Parallel and Distributed Computing.*

[7] B. T. N. Gunney. Scalable mesh management for patch-based amr, 2012. Nuclear Explosive Code Development Conference Livermore, CA, United States.

[8] S.Lee K.W.Cho. Parallel approach of fully systemized chimera methodology for steady/unsteady problems. *Parallel Computational Fluid Dynamics 2002*, 2002.

[9] D. T. Graves J. N. Johnson H. S. Johansen N. D. Keen T. J. Ligocki D. F. Martin P. W. McCorquodale D. Modiano P. O. Schwartz T. D. Sternberg B. Van Straalen M. Adams, P. Colella. Chombo software package for amr applications design document, 2015.

[10] Terry J. Ligocki vLeonid Oliker John Shalf Brian Van Straalen Samuel W. Williams Matthias Christen, Noel Keen. Automatic thread-level parallelization in the chombo amr library, 2011.

[11] Robert L. Meakin. Composite overset structured grids, handbook of grid generation, chapter 11. *CRC Press.*

[12] T. J. Ligocki D. Modiano B. Van Straalen P. Colella, D. T. Graves. Ebchombo software package for cartesian grid, embedded boundary applications, 2003.

[13] N. Anders Petersson. Hole-cutting for three-dimensional overlapping grids. *SIAM Journal on Scientific Computing.*

[14] Scott R. Kohn Richard D. Hornung. Managing application complexity in the samrai object oriented framework, 2002.

[15] Peter Zinterhof Roman Trobec, Marian Vajtersic. *Parallel Computing - Numerics, Applications, and Trends.* Springer, 2008.

[16] D. B. Gannon M. L. Norman S. B. Baden, N. P. Chrisochoides. *Structured Adaptive Mesh Refinement (SAMR) Grid Methods.* Springer, 2000.

[17] Norman E. Suhs, Stuart E. Rogers, and W. E. Dietz. Pegasus 5: An automatic pre-processor for overset-grid cfd. AIAA 32nd Fluid Dynamics Conference, St. Louis.

[18] R. D. Hornung S. R.Kohn. S. S. Smith N. S. Elliott Wissink, A. M. Large scale structured amr calculations using the samrai framework, 2001. SC01 Proceedings, Denver.

Chapter 2

Python

ABC	A Programming Language
CWI	Centrum Wiskunde & Informatica
VM	Virtual Machine
VHLL	Very High Level Language
CLR	Common Language Runtime
JIT	Just in Time
JVM	Java Virtual Machine
PSF	Python Software Foundation
GIL	Global Interpreter Lock
MPI	Message Passing Interface
ALU	Arithmetic Logic Unit

Python is an interpreted high-level programming language for general-purpose programming, which provides constructs that enable clear coding on both small and large scales, thanks to its dynamic type system and automatic memory management.

Created by Guido van Rossum (Figure 2.1a) and first released in 1991, Python has a design philosophy that emphasizes code readability, and a

syntax that allows programmers to express concepts in fewer lines of code than with other languages.
It supports multiple programming paradigms[1], including object-oriented, functional[2] and procedural, and has a large and comprehensive standard library.

Python interpreters are available for many operating systems. CPython, the reference implementation of Python, is an open source software developed using a community-based model, as do nearly all of its variants.

In this chapter we will introduce this versatile and multipurpose programming language, taking inspiration from who is really into the Python programming pattern and philosophy (in my humble opinion, I am just a novice who loves to learn) like, just to name a few, *Alex Martelli*, *Brett Slatkin*, *Kurt Smith*, *Lisandro Dalcin.* Some of their works are reported in bibliography.

2.1 A Bit of History

What do the alphabet and the programming language Python have in common? Yes, they both start with ABC. And if we are talking about ABC in the Python context, it's clear that we mean the not so famous programming language[3].

Python was conceptualized in the late 1980s. In an interview, Guido van Rossum said: "In the early 1980s, I worked in a team building a language called ABC. I don't know how well people know ABC's influence on Python, but I try to mention it because I'm indebted to everything I learned during that project and to the people who worked on it."

Later on in the same interview, Guido van Rossum continued: "(...) I

[1]Styles of building the structure and the elements of computer programs.

[2]In computer science, functional programming treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, so programming is done with expressions or declarations instead of statements. Here, the output value of a function depends only on its input arguments. So, calling a function f twice with the same value for an argument x produces the same result *f(x)* each time, not depending on a local or global scope.

[3]ABC is a general-purpose programming language and programming environment, which had been developed at CWI, Netherlands.

started typing, and I created a simple virtual machine[4], a simple parser[5], and a simple runtime[6]. I made my own version of the various ABC features that I liked, creating a basic syntax, using indentation for statement grouping instead of curly braces or begin-end blocks, and developing a small number of powerful data types: hash tables[7] (or dictionaries, as we call it), lists, strings, and numbers."

But what about the name "Python". Most people think about snakes, spotting also the logo (see Figure 2.1b), but the origin of the name has its root in British humour.

Guido van Rossum wrote in 1996 about the choice of the name: "Over six years ago, in December 1989, I was looking for a 'hobby' programming project that would keep me occupied during the week around Christmas. My office (...) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)."

2.2 Releases

The first version of Python was released in February 1991. This was version 0.9.0, and the release included already exception handling, functions, core data types, a module system and an object oriented support.

Python version 1.0 was released in January 1994, including as new features

[4]In computing, a virtual machine (VM) is an emulation of a computer system, providing its functionalities. Their implementations may involve specialized hardware, software, or a combination of both.

[5]Within computational linguistics, the term *parsing* is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents.

[6]A runtime system, also called *run-time system*, primarily implements portions of an execution model. Most languages have some form of runtime system, which implements control over the order in which work that was specified in terms of the language gets performed.

[7]In computing, a hash table (*hash map*) is a data structure which implements an associative array abstract data type, a structure that can map keys to values. Given a key, a hash table uses a hash function to compute an index into an array of buckets (or slots), from which the desired value can be found.

(a) Guido Van Rossum

(b) Python's logo

Figure 2.1: Python: its creator and its logo

the functional programming tools: lambda, map, filter, reduce.

Python 2.0 was released in October 2000 and had many major new features, including a cycle-detecting garbage collector[8] and support for Unicode[9]. With this release, the development process became more transparent and community-backed.

Python 3.0 was released in December 2008 after a long testing period. It is a major revision of the language that is not backward-compatible with previous versions. However, many of its major features have been backported to the backward-compatible Python 2.6.x and 2.7.x version series. The emphasis in this release had been on the removal of duplicate programming constructs and modules, thus fulfilling (or coming close to) the 13th postulate of the Zen of Python[10], whose aphorisms are reported in the hereunder section.

2.3 The Zen of Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.

[8]The gc exists only to detect and free circular references. Non-circular references are handled through refcounting.

[9]Unicode is a computing industry standard for the consistent encoding, representation, and handling of text, expressed in most of the world's writing systems.

[10]The Zen of Python is a collection of 20 software principles (only 19 of which have been written down) that influences the design of Python Programming Language. Tim Peters, long time Pythoneer, envisioned it.

- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one –obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

The twentieth aphorism, according to Internet, seems to be some bizarre Tim Peters in-joke: it's an opportunity for people to provide their own addition. And this "artistic license" enlightens considerably the *free* ("as in speech" or "as in beer") spirit of Python.

2.4 The Python Language

The Python language, while not minimalist, is rather spare, for good pragmatic reasons: a complicated language is harder to learn and master (and to implement efficiently and without bugs) than a simpler one.
Complications and quirks in a language hamper productivity in software development and maintenance, particularly in large projects, where many developers cooperate and often maintain code originally written by others.

Python is simple, but not simplistic. It adheres to the idea that if a language behaves a certain way in some contexts, it should ideally work similarly in all contexts (this philosophy recalls a bit the following principle of the *Duck typing*[11]: "If it walks like a duck and it quacks like a duck, then it must be a duck"). A good language, like any other well-designed artifact, must balance such general principle with a high degree of practicality.

Python is a general-purpose programming language: its traits are useful in just about any area of software development, but it does not have to stand alone. In fact, while many developers find that it fills all of their needs, Python programs should cooperate with a variety of other software components, making it an ideal language for gluing together components written in other languages.

Python is a very-high-level language (VHLL[12]), which means that it uses a higher level of abstraction, conceptually farther from the underlying machine, than do classic compiled languages such as C, C++, and Fortran, which are traditionally called high-level languages.

Python is also simpler, faster to process (both for human brains and for programmatic tools), and more regular, than classic high-level languages. This enables high programmer productivity and makes Python an attractive development tool.

Good compilers for classic compiled languages can generate binary machine code that runs faster than Python code, but in most cases, the performance of Python-coded applications is sufficient.
However, when it doesn't, optimization techniques can be used to improve program's performance, while keeping the benefit of high productivity.

Newer languages such as Java are slightly higher-level than classic ones (such as C and Fortran), and share some characteristics of classic languages (for example, the need to use declarations) as well as some of VHLLs like Python

[11]In computer programming, duck typing requires that type checking be deferred to runtime. With normal typing, suitability is assumed to be determined by an object's type only. In duck typing, an object's suitability is determined by the presence of certain methods and properties, rather than the actual type of the object.

[12]VHLLs are designed to reduce the complexity and amount of source code required to create a program, incorporating higher data and control abstraction abilities.

(to name but a few, the use of portable bytecode[13] as the compilation target in typical implementations, and garbage collection to relieve programmers from the need to manage memory).

Python is an object-oriented programming language, but lets the developer to code using different styles, mixing and matching as the application requires. Its object-oriented features are conceptually similar to those of C++, but simpler to use. And, at the same time, more deeper: **everything**, in Python, is an object[14].

2.5 The Standard Python Library

The standard Python library is nearly as important for effective Python use as the language itself. In fact, it supplies many well-designed, solid, 100 percent pure Python modules for convenient reuse, working on all its supported platforms (extension modules that are not coded in Python, however, do not necessarily enjoy the same automatic cross-platform portability as pure Python code).

Extension modules, from the standard library or from elsewhere, let Python code access functionality supplied by the underlying operating system or other software components such as GUIs[15], databases, and networks.

Extensions also afford maximal speed in computationally intensive tasks such as XML parsing and numeric array computations.
Users can write special-purpose extension modules in lower-level languages to achieve maximum performance for small, computationally intensive parts that they originally prototyped in Python.
They can also use tools such as Cython to wrap existing C/C++ libraries into Python extension modules (we will see it).

Finally, developer can embed Python in applications coded in other languages, exposing existing application functionality to Python scripts via

[13]Bytecode, also termed *portable code* or *p-code*, is a form of instruction set designed for efficient execution by a software interpreter.

[14]From [8] (free book): "everything in Python is an object, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the doc string defined in the function's source code. The *sys* module is an object which has (among other things) an attribute called path. And so forth."

[15]Graphical user interfaces.

dedicated Python extension modules.

2.6 Python Implementations

When people speak of Python, they often mean not just the language but also its implementation. Python is actually a specification for a language that can be implemented in many different ways, emphasising the distinction between the language and its implementations.
Here follows a list of the most common Python implementations, on the understanding that this thesis will cover just the first one.

- CPython is Classic Python and is the most up-to-date, solid and complete production-quality implementation of Python. It can be considered the “reference implementation” of the language.
  CPython is a compiler, an interpreter, and a set of built-in and optional extension modules, all coded in standard C.
  CPython can be used on any platform where the C compiler complies with the ISO/IEC 9899:1990 standard[16].

- Jython is a Python implementation for any Java Virtual Machine (JVM) compliant with Java 7 or better. For optimal use of Jython, users need some familiarity with fundamental Java classes and its supporting tools for tasks such as manipulating .jar files.

- IronPython is a Python implementation for the most commonly known .NET, which make it possible to use all CLR[17] libraries and frameworks, having of course some familiarity with them.

- PyPy is a fast and flexible implementation of Python, coded in a subset of Python itself, able to target several lower-level languages and virtual machines using advanced techniques such as type inferencing[18].
  PyPy’s greatest strength is its ability to generate native machine code “just in time”[19].

[16]The international standard which defines the C programming language is available from ISO. The current revision is ISO/IEC 9899:2011, also called C11.

[17]Microsoft-designed Common Language Runtime (CLR).

[18]Type inference refers to the automatic detection of the data type of an expression in a programming language.

[19]In computing, just-in-time (JIT) compilation is a way of executing computer code that involves compilation during execution of a program rather than prior to execution. Most often this consists of source code or more commonly bytecode translation to machine code, which is then executed directly.

The primary difference between the implementations is the environment in which they run and the libraries and frameworks they can use:

> if you need a JVM environment, then Jython is your choice;
>
> if you need a CLR (a.k.a. .NET) environment, take advantage of IronPython;
>
> if you need a custom version of Python, consider PyPy;
>
> if you're mainly working in a traditional environment, CPython is an excellent fit, offering a most widely support by third-party addons and extensions.

In other words, when you're experimenting, learning, and trying things out, use CPython, most widely supported and mature. To develop and deploy, your best choice depends on the extension modules you want to use and how you want to distribute your programs.

2.7 Python's Pros and Cons

2.7.1 Pros...

Python is billed by the Python Software Foundation[20]. Launched on March 6, 2001, its mission is to foster development of the Python community being responsible for various processes within it, like developing the core Python distribution, managing intellectual rights, organizing developer conferences, and so on.
Being easy to learn and running everywhere, in fact, implies having a huge pool of consumers to follow and to stage.

Python it's useful for a range of application types (including Web development, scientific computing, education, etc.). And thanks to its usefulness, the language scores well in popularity indexes.

Read it, use it with ease. The main characteristics of a Python program is that it is easy to read, and this aspect has many benefits: it helps the programmer to think more clearly when writing programs, and it helps the others who will maintain or enhance your program. In both cases, it

[20]The Python Software Foundation (PSF) is a nonprofit organization devoted to the Python programming language.

requires less effort to write a Python program than to write one in another language like C++ or Java, facilitating open source development.

Python is easy to use. Thanks to its learning fast curve, and to its being dynamically typed[21] and flexible, with code that is less verbose, it has become extremely popular in academia, creating a large talent pool.

Python is a very productive way to write code. Some of this comes from the simple syntax and readability (there is virtually no boilerplate[22]). Some of it comes from the rich, well-designed built-ins, from the standard library and from the availability of many third-party open source libraries and modules.
Being easy to understand, the code is easier to maintain, of course.

Internet of things opportunities. Python may become popular for the Internet of things, as new platforms such as Raspberry Pi[23] are based on it. Raspberry Pi's documentation cites the language as "a wonderful and powerful programming language that's easy to use (easy to read and write), and with Raspberry Pi lets you connect your project to the real world."

Asynchronous coding benefits. Synchronous programming code, excluding conditionals and function calls, is executed sequentially from top-to-bottom, blocking on long-running tasks such as network requests and disk I/O.
Asynchronous programming[24], on the contrary, runs in an event loop, and when a blocking operation is needed, the request is started, and the code keeps running without blocking for the result.
If the response is ready, an interrupt is fired, which causes an event handler

[21]A language is statically typed if the type of a variable is known at compile time. On the other hand a language is dynamically typed if the variable's type is associated with run-time values.

[22]In computer programming, boilerplate code (or *boilerplate*) refers to sections of code that have to be included in many places with little or no alteration. It is often used when referring to languages that are considered verbose, i.e. the programmer must write a lot of code to do minimal jobs at all.

[23]The Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote the teaching of basic computer science in schools and in developing countries. It does not include peripherals (such as keyboards, mice and cases); however, some accessories have been included in several official and unofficial bundles.

[24]User interfaces, for example, are asynchronous by nature, and spend most of their time waiting for user input to interrupt the event loop and trigger event handlers.

to be run, letting the control flow to continue.
In this way, a single program thread can handle many concurrent operations.

Python is great for writing asynchronous code, which rather than threading uses a single event loop to do work in small units.
This code is often easier to write and maintain without confusing resource contention[25], deadlocks[26], and other synchronisation problems.

In Python, everything is an object. Although this intrinsic attitude make it possible to write very clear and understandable object-oriented code, Python does not oblige developers to create an OO class mandatorily, to start programming (like instead Java does).
Python, therefore, truly support multiparadigms approaches.

Python has a huge community.One of the greatest strengths of Python is its robust, friendly, welcoming, international community. Python programmers and contributors meet face to face at conferences and local user groups, actively discuss shared interests, and help one another on mailing lists and social media.

2.7.2 ...and Cons

Speed can be an issue. Because it is an interpreted language, it is often many times slower than compiled languages. However, it comes back to separating the language from the runtime: certain benchmarks of Python code conducted under PyPy, run faster than the equivalent C code or others.
To cope with its slow speed of execution, many Python packages have been optimized over the years and execute at compiled speed, although, in principle, performance remain slower compared to older languages such as C/C++ (in this regard, we will see in the next section a tool that comes to the aid of Python).

Design restrictions. Because the language is dynamically typed, it requires more testing and has errors that only show up at runtime.

[25]In computer science, resource contention is a conflict over access to a shared resource such as random access memory, disk storage, cache memory, internal buses or external network devices.

[26]In concurrent computing, a deadlock is a state in which each member of a group is waiting for some other member to take action, such as sending a message or more commonly releasing a lock.

Python's global interpreter lock (read section 2.10), or *GIL*, is another restriction and it means that only one thread can access Python internals at a time. This may be less important these days, since you can so easily spawn tasks out to separate processes using the multiprocessing module, or write asynchronous code instead, but it remains a big deal (surmountable, nervertheless).

Significant whitespace is enforced by the interpreter. The structure of Python programs must be **consistent**, so where brackets or other identifiers allow the user more freedom in other languages, indentation is what matters when it comes to Python.

Absence from mobile computing and browsers. Python is present on many server and desktop platforms, but it is weak in mobile computing; very few smartphone applications are developed with Python, and it is also rarely seen on the client side of a Web application.
Python isn't in Web browsers (and that's a real shame), being hard to secure. There is Brython[27], but maybe it's not real-world usable.

2.8 Cython

Cython is a superset of the Python programming language, designed to give C-like performance with code which is mostly written in Python.
It is a compiled language that generates CPython extension modules, which can then be loaded and used by regular Python code using the import statement.

Cython is a derivative of the Pyrex[28] language, but supports more features and optimizations than it does, and works on Windows, macOS, and Linux, producing source files compatible with CPython 2.6, 2.7, and 3.3 through 3.7.

Pyrex was developed by Greg Ewing, and its ability to speed up Python code by large factors made it instantaneously popular, and many projects

[27]Brython, or Browser Python (it's also the Welsh word for "breton"), is designed to replace Javascript as the scripting language for the Web. As such, it is a Python 3 implementation adapted to the HTML5 environment.

[28]Pyrex is a Python-like language for rapidly and easily writing python extension modules. It can be described as Python with C data types.

started using it intensively, especially for the scientific Python community (needing some performance improvements), although it did not intend to support all constructs in the Python language.
Being a successful open source concept, other projects adapted and patched it to fit their needs, and among those, two forks of Pyrex itself(one by Stefan Behnel and the other one, Sage[29], by William Stein), ultimately combined to form the Cython project, under the leadership and guidance of Robert Bradshaw and Stefan Behnel.

From its beginning, Cython has benefited from a large and active open source community, with many contributions to develop new features, to implement them, to report bugs, and to fix them.

2.9 What Cython is

Cython is often confused with CPython, but the two are very different.

CPython, in fact, provides a C-level interface into the Python language, known as the Python/C API[30], while Cython, on the other side, uses this C interface extensively, and therefore Cython depends on CPython.
So, Cython is not another implementation of Python, and indeed it needs the CPython runtime to run the extension modules it generates.

But then, what exactly is Cython?

Cython is two closely related things:

- Cython is a programming language that blends Python with the static type system of C and C++.
- Cython is a compiler that translates Cython source code into efficient C or C++ source code. This source can then be compiled into a Python extension module or a standalone executable.

[29]Sage is a GPL-licensed comprehensive mathematics software system that aims to provide a viable alternative to Maple, Mathematica, and Matlab.
[30]In computer programming, an application programming interface (API) is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components.

Cython's potential comes from the way it combines Python and C: writing code feeling like Python, but getting easy access to C power. So, it can be considered halfway between high-level Python and low-level C. And why mixing such different languages? Let see.

All the positive aspects which have revealed Python to the vast programmers' world (high-level, dynamic, easy to learn, flexible, etc.) come with a cost: being interpreted. And that cost means a possible slowness of several orders of magnitude than statically typed compiled languages.

C, on the other hand, is one of the oldest statically typed compiled languages in widespread use, so compilers have had time and study cases to optimize its performance.
C is very low level and very powerful, but unlike Python, it does not have many safeguards in place and can be difficult to use.

So, now it starts to be clear the answer to the previous question "why combine these two languages": both of them are mainstream, but they are typically used in different domains, given their differences. And Cython's beauty is this: it combines Python's expressiveness and dynamism with C's bare-metal performance, allowing its compiler to generate efficient C code, just using a small number of keywords to the Python language.

So, a typical approach of a developer could be starting with Python code, easy to write and to debug.
After that, needing better performances, profiling the code to search the parts to be optimized.
Lastly, to speed up Python code, compiling Python source with Cython, with optional static type declarations to achieve performance improvements, depending of course on the algorithm to maximise.

But a developer might want to take the inverse approach, starting with a C or C++ library and to interface it with Python. On this matter, he can use Cython to interface with external code and create optimized wrappers (we will see this later).

Both capabilities (compiling Python and interfacing with external code) are designed to work together well, and each is an essential part of what makes Cython useful. With Cython, either direction is feasible, coming from either starting point.

2.10 Cython Features

From the beginning, Cython has had an ambitious goal, that is full Python compatibility. But it has also acquired features that are specific to its unique nature between Python and C, giving him the fact of being a real Python superset, easy to use and more efficient.
Facilities worthy of note are:

- easier interoperability and conversion between C types and Python types.

- Specialized syntax to ease wrapping and interfacing with C++.

- Automatic static type inference for certain code paths.
  All C types, infact, are available for type declarations: integer and floating point types, complex numbers, structs, unions and pointer types. Cython can automatically and correctly convert between the types on assignment.
  This also includes Python's arbitrary size integer types, where value overflows on conversion to a C type will raise a Python *OverflowError* at runtime.

- First-class buffer support with buffer-specific syntax: using buffers effectively is often the key to obtaining C-level performance from Cython code.
  Cython makes particularly easy to work with them, having full support for the new buffer protocol and, with it, NumPy arrays.

- Typed memoryviews. As suggested by the name, a typed memoryview is used to view (or better, to share) data from a buffer-producing object, and overlaps with the Python memoryview[31] type, expanding it. Because a typed memoryview operates at the C level, it has minimal Python overhead and is very efficient, but it has a memoryview-like interface, so it is easier to use than working with C-level buffers directly.
  And being designed to work with the buffer protocol, it supports any buffer-producing object efficiently, allowing sharing of data buffers without copying.

[31]Memoryview is a Python type whose sole purpose is to represent a C-level buffer at the Python level.

- Thread-based parallelism with *prange*. In the section dedicated to the pros and cons of Python, we have introduced the global interpreter lock, or GIL.
  According to Python's documentation, the GIL is "a mutex[32] that prevents multiple native threads from executing Python bytecodes at once."
  In other words, the GIL ensures that only one native (or OS-level) thread executes Python bytecodes at any given time during the execution of a CPython program, affecting not just Python-level code, but the Python/C API as a whole.

  Why is the GIL necessary? "This lock is necessary mainly because CPython's memory management is not thread-safe", Python's documentation explains. However, there is nothing so bad that it is not good for something. After all:

  1) C code not working with Python objects can be run without the GIL in effect, allowing fully threaded execution.
  2) The GIL is specific to CPython and consequently, other Python implementations like Jython, IronPython, and PyPy, have no need for a GIL.
  3) Although being controversial because it prevents multithreaded CPython programs from taking full advantage of multiprocessor systems, long-running operations such as I/O, image processing, and *NumPy*[33] number crunching, happen outside the GIL.
  4) Developers can also use *multiprocessing* module to run applications on different processes, rather than on multiple threads, as we previously said. And being in this context, another solution is to use *MPI4Py*[34], as we will see later in the thesis, to use message passing[35]

[32]In computer science, mutual exclusion is instituted for the purpose of preventing race conditions; it is the requirement that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section.

[33]NumPy is the fundamental package for scientific computing with Python.

[34]MPI for Python package. MPI for Python provides bindings of the Message Passing Interface (MPI) standard for the Python programming language, allowing any Python program to exploit multiple processors.

[35]In computer science, message passing sends a message to a process and relies on the process and the supporting infrastructure to select and invoke the actual code to run. Message passing differs from conventional programming where a process, subroutine, or function is directly invoked by name. Message passing is the key to some models of concurrency and object-oriented programming.

> among parallel computational nodes.
>
> Because Cython code is compiled, not interpreted, it is not running Python bytecode. Hence, getting the chance to create C-only entities in Cython that are not tied to any Python object, we can release the global interpreter lock when working with the C-only parts of Cython.
>
> Put in another way, we can use Cython to bypass the GIL and achieve thread-based parallelism: Python users' dream!

In the next two sections we will discuss some of its features previously listed; no GIL exaples will be shown, because it is not our main objective. Instead, a comparison between Python, C and Cython function will be discussed, referring performances and timings.
The discussions are taken from the two homonymous sections of [11].

2.11 Comparing Python, C, and Cython

Let's see an example to understand the differences between these three languages.

Consider a simple Python function *fib* that computes the nth Fibonacci number:

```python
def fib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

The C transcription of *fib* follows the Python version closely:

```c
double cfib(int n) {
    int i;
    double a=0.0, b=1.0, tmp;
    for (i=0; i<n; ++i) {
        tmp = a; a = a + b; b = tmp;
    }
    return a;
}
```

We use *doubles* in the C version and *floats*[36] in the Python version to make the comparison direct and remove any issues related to integer overflow for C integral data types.

Cython understands Python code, so our unmodified Python *fib* function is also valid Cython code.
To convert the dynamically typed Python version to the statically typed Cython version, we use the **cdef** Cython statement to declare the statically typed C variables *i*, *a*, and *b*:

```
def fib(int n):
    cdef int i
    cdef double a=0.0, b=1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

What about performances? Table 2.1 has the results.

Version	fib(0) [ns]	Speedup	fib(90) [ns]	Speedup	Loop body [ns]	Speedup
Pure Python	590	1	12.852	1	12.262	1
Pure C	2	295	164	78	162	76
C extension	220	3	386	33	166	74
Cython	90	7	258	50	168	73

Table 2.1: Fibonacci timings for different implementations

In Table 2.1, the second column measures the runtime for *fib(0)* and the third column measures the speedup of *fib(0)* relative to Python.
Because the argument to *fib* controls the number of loop iterations, *fib(0)* does not enter the Fibonacci loop, so its runtime is a reasonable measure of the language-runtime and function-call overhead.
The fourth and fifth columns measure the runtime and speedup for *fib(90)*, which executes the loop 90 times. Both the call overhead and the loop execution runtime contribute to its runtime.
Lastly, the sixth and seventh columns measure the difference between the *fib(90)* runtime and the *fib(0)* runtime and the relative speedup; difference that is an approximation of the runtime for the loop alone, having removed runtime and call overhead.

[36]Python's built-in *float* type has *double* precision (it's a C *double* in CPython).

But beyond the columns, table 2.1 has also four rows:

- Pure Python. The first row (after the header) measures the performance of the pure Python version of *fib* , and as expected, it has the poorest performance by a significant margin in all categories.
  In particular, the call overhead for *fib(0)* is over half a microsecond on this system. Each loop iteration in *fib(90)* requires nearly 150 nanoseconds; Python leaves much room for improvement.

- Pure C. The second row measures the performance of the pure C version of *fib*. Here there is no interaction with the Python runtime, so there is minimal call overhead. This also means it cannot be used from Python.
  This version provides a bound for the best performance we can reasonably expect from a simple serial *fib* function.
  The *fib(0)* value indicates that C function call overhead is minimal (2 nanoseconds) when compared to Python, and the *fib(90)* runtime (164 nanoseconds) is nearly 80 times faster than Python's on this particular system.

- Hand-written C extension. The third row measures a hand-written C extension module for Python 2. This extension module requires several dozen lines of C code, most of it boilerplate that calls the Python/C API.
  When calling from Python, the extension module must convert Python objects to C data, compute the Fibonacci number in C, and convert the result back to a Python object. Its call overhead (the *fib(0)* column) is correspondingly larger than that of the pure-C version, which does not have to convert from and to Python objects.
  Because it is written in C, it is about three times faster than pure Python for *fib(0)*. It also gives a nice factor-of-30 speedup for *fib(90)*.

- Cython. The last row measures the performance for the Cython version.
  Like the C extension, it is usable from Python, so it must convert Python objects to C data before it can compute the Fibonacci number, and then convert the result back to Python.
  Because of this overhead, it cannot match the pure C version for *fib(0)*, but, notably, it has about 2.5 times less overhead than the hand-written C extension.

Because of this reduced call overhead, it is able to provide a speedup of about a factor of 50 over pure Python for *fib(90)*.

The takeaways from Table 2.1 are the last two columns: the loop runtime for the pure C, C extension, and Cython versions are all about 165 nanoseconds, and the speedups relative to Python are all approximately 75x (for the C-only parts of an algorithm, provided sufficient static type informations, Cython can usually generate code that is as efficient as a pure-C equivalent).

So, when properly accounting for Python overhead, we see that Cython achieves C-level performance. Moreover, it does better than the hand-written C extension module on the Python-to-C conversions. In fact, Cython generates highly optimized code that is frequently faster than an equivalent hand-written C extension module. It is often able to generate Python-to-C conversion code that is several factors faster than naive calls into the Python/C API.

We can go even further and use Cython to create Python-like C functions that have no Python overhead.
These functions can be called from other Cython code but cannot be called directly from Python. They allow us to remove expensive call overhead for core computations.

What is the reason for Cython's performance improvements? For this example, the likely causes are function call overhead, looping, math operations, and stack versus heap allocations[37].

- **Function Call Overhad**. The *fib(0)* runtime is mostly consumed by the time it takes to call a function in the respective language; the time to run the function's body is relatively small.
  We see in Table 2.1 that Cython generates code that is nearly an order of magnitude faster than calling a Python function, and more than two times faster than the hand-written extension.
  Cython accomplishes this by generating highly optimized C code that bypasses some of the slower Python/C API calls. We use these API calls in the preceding C-extension timings.
- **Looping**. Python for loops, as compared to compiled languages, are notoriously slow.

[37]The stack is the memory set aside as scratch space for a thread of execution, while the heap is memory set aside for dynamic allocation.

One surefire way to speed up loopy Python code is to find ways to move the Python for and while loops into compiled code, either by calling built-in functions or by using something like Cython to do the transformation for you. The *fib(90)* column in the table is running a for loop in each language for 90 iterations, and we see the impact of this operation on the different version runtimes.

- **Math Operations**. Because Python is dynamically typed and cannot make any type-based optimizations, an expression like a + b could do anything.
  We may know that a and b are only ever going to be floating-point numbers, but Python never makes that assumption. So, at runtime, Python has to look up the types of both a and b (which, in this instance, are the same).
  It must then find the type's underlying *__add__* method (or the equivalent), and call *__add__* with a and b as arguments. Inside this method, the Python *float*s a and b have to be unboxed to extract the underlying C *double*s, and only then can the actual addition occur.
  The result of this addition has to be packaged in an entirely new Python *float* and returned as the result.
  The C and Cython versions already know that a and b are *double*s and can never be anything else, so adding a and b compiles to just one machine code instruction.

- **Stack vs Heap Allocation**. At the C level, a dynamic Python object is entirely heap allocated.
  Python takes great pains to intelligently manage memory, using memory pools and internalizing frequently used integers and strings.
  But the fact remains that creating and destroying objects (and objects here means ANY objects, even scalars) incurs overhead to work with dynamically allocated memory and Python's memory subsystem.
  Because Python float objects are immutable, operations using Python *float*s involve the creation and destruction of heap-allocated objects.
  The Cython version of *fib* declares all variables to be stack-allocated C *double*s.
  As a rule, stack allocation is much faster than heap allocation. Moreover, C floating-point numbers are mutable, meaning that the for loop body is much more efficient in terms of allocations and memory usage.
  It is not surprising that the C and Cython versions are more than an order of magnitude faster than pure Python, since the Python loop

body has to do so much more work per iteration.

2.12 Wrapping C Code with Cython

Let's consider Cython's other main feature: interfacing with external code. Suppose that, instead of Python code, our starting point is C or C++ code, and that we want to create Python wrappers for it. Because Cython understands C and C++ declarations and can interface with external libraries, and because it generates highly optimized code, it is easy to write efficient wrappers with it.

Continuing with our Fibonacci theme, let's start with a C implementation and wrap it in Python using Cython. The interface for our function is:

```
double cfib(int n);
```

while the Cython wrapper results in:

```
cdef extern from "cfib.h":
    double cfib(int n)

def fib(n):
    """Returns the nth Fibonacci number."""
    return cfib(n)
```

The **cdef** extern block provides the *cfib.h* header filename, and after that, a *fib* Python wrapper function is defined, which calls *cfib* and returns its result.

After compiling the preceding Cython code into an extension module named *wrap_fib*, it can be used from Python directly:

```
>>> from wrap_fib import fib
>>> help(fib)
Help on built-in function fib in module wrap_fib:
fib(...)
    Returns the nth Fibonacci number.
>>> fib(90)
2.880067194370816e+18
```

The *fib* function is a regular Python function inside the *wrap_fib* extension module, and calling it with a Python integer does what we expect, calling into the underlying C function for us and returning a (large) result.

A hand-written wrapper would require several dozen lines of C code, and detailed knowledge of the Python/C API, while here it was just a handful of lines of Cython code to wrap a simple function.

This example was intentionally simple: provided the values are in range, a Python *int* converts to a C *int* without issue, raising an *OverflowError* otherwise.
Internally the Python *float* type stores its value in a C *double*, so there are no conversion issues for the *cfib* return type.
And because we are using simple scalar data, Cython can generate the type conversion code automatically.

However, being Cython a full-fledged language, we can use it to do whatever we like before and after the wrapped function call.
Because the Cython language understands Python and has access to Python's standard library, we can leverage all of Python's power and flexibility.

It should be noted that we can use Cython's two *reasons for being* in one file (or doing both inside the same function): speeding up Python alongside calling external C functions.

2.13 The Pareto Principle

Of course it can be exiting to see improvements when we add some trivial **cdef** statements to Python code, but not all Python code will see performance gains when compiled with Cython.

The preceding *fib* example is intentionally CPU bound, meaning that all the runtime is spent manipulating a few variables inside CPU registers, and little, if any, movement is required.

If this function were, instead, memory bound (like adding the elements of two large arrays), I/O bound (e.g., reading a large file from disk), or network bound (downloading a file from an FTP server), the performance difference between Python, C, and Cython would likely be significantly decreased. And as said, I/O and Numpy number crunching operations happen outside the GIL, so parallel execution can be exploited intrinsically in Python itself.

When improving Python's performance is the goal, the Pareto principle[38] works in our favor: we can expect that approximately 80 percent of a program's runtime is due to only 20 percent of the code.

Of course the 20 percent is very difficult to locate without profiling, but there is no excuse not to profile Python code, given how simple its built-in profiling tools are to use.
Modules *cProfile* and *profile*[39] provide deterministic profiling of Python programs. Deterministic profiling is meant to reflect the fact that all function call, function return, and exception events are monitored, and precise timings are made for the intervals between these events (during which time the user's code is executing).
In contrast, statistical profiling randomly samples the effective instruction pointer, and deduces where time is being spent.
The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

That said, if we determine via profiling that the bottleneck in our program is due to it being I/O or network bound, then we cannot expect Cython to provide a significant improvement in performance, and we can use instead Python modules like *multiprocessing* and *threading*, to achieve good results with less effort.
it is worth determining the kind of performance bottleneck you have, before turning to Cython, because, although it is a powerful tool, Cython must be used in the right way and in the right context.

And once started to use it, all limitations of C data types become relevant concerns, because Cython brings C's type system to Python.
For example, Python integer objects silently convert to unlimited-precision[40]

[38]The Pareto principle (also known as *the 80/20 rule*, *the law of the vital few*, or *the principle of factor sparsity*) states that, for many events, roughly 80% of the effects come from 20% of the causes.

[39]*cProfile* is a C extension with reasonable overhead; *profile* is a pure Python module, whose interface is limitated by cProfile, but which adds significant overhead to profiled programs

[40]In computer science, arbitrary-precision arithmetic (also called *bignum arithmetic*, *multiple-precision arithmetic*, or sometimes *infinite-precision arithmetic*), indicates that

Python long objects when computing large values, but C *int*s or *long*s are fixed precision, meaning that they cannot properly represent unlimited-precision integers.

Cython has features to help catch these overflows, however the larger point remains: C data types are faster than their Python counterparts, but are sometimes not as flexible or general.

2.14 Summary

This chapter has introduced the Python language, emphasising its essential features and pointing out its pros and cons.
Its drawbacks coding CPU bound and multi-processes/multi-threads applications were pointed out, and clarifications and solutions to these downsides have been said, by focusing particularly on Cython, a performing superset of Python, which can be the amalgam between its subset and the HPC ecosystem.
The evidence of that are its continuing use in all those Python modules written ad hoc for scientific computing (introduced in paragraph 1.4), and that software and hardware giants (like Intel, NVIDIA, universities and private enterprise) are pressing in this direction.

Bibliography

[1] Steve Holden Alex Martelli, Anna Ravenscroft. *Python in a Nuthshell - the Definitive Reference.* O'Reilly, 2016.

[2] Allen B. Downey. *Think Python - How to Think Like a Computer Scientist.* O'Reilly, 2015.

[3] Mark Lutz. *Learning Python: Powerful Object-Oriented Programming.* O'Reilly, 2013.

[4] Alex Martelli. The template method design pattern in python, 2003. OSCON.

[5] Alex Martelli. Thread in python 2.3, 2003. OSCON.

calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system. This contrasts with the faster fixed-precision arithmetic found in most arithmetic logic unit (ALU) hardware, which typically offers between 8 and 64 bits of precision.

[6] Alex Martelli. Template method and factory desing patterns, 2003. EuroPython.

[7] Ian Ozsvald Micha Gorelick. *High Performance Python: Practical Performant Programming for Humans*. O'Reilly, 2014.

[8] Mark Pilgrim. *Dive Into Python*. Apress, 2011.

[9] Luciano Ramalho. *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly, 2015.

[10] Brett Slatkin. *Effective Python: 59 Specific Ways to Write Better Python (Effective Software Development Series)*. Addison-Wesley Professional, 2015.

[11] Kurt W. Smith. *Cython - a Guide for Python Programmers*. O'Reilly, 2015.

Chapter 3

Quadtrees and Octrees

ADT	Abstract Data Type
DF	Depth First
DFS	Depth First Search
CFD	Computational Fluid Dynamics
PABLO	Parallel Balanced Linear Octree
PABLitO	Parallel Balanced Linear interpreted Octree

In this chapter, an overview of hierarchical data structures for representing images, such as the quadtree and octree, is presented.

Hierarchical data structures are necessary when the scene being modeled becomes significantly larger than the display grid, arising major logistic problems.
In this case, there are two ways two handle these problems: one approach is based on object-space hierarchies, while the second one is centered on image-space hierarchies, and it is typified precisely by hierarchical data structures such as quadtrees and octrees. We will give in the next section a more precise clarification.

Octrees are the *3D* analog of quadtrees. The name is formed from *oct* and *tree*, but note that it is normally written "octree" with only one "t". Octrees are often used in 3D graphics and 3D game engines, and their use for

computer graphics was pioneered[1] by Donald Meagher at Rensselaer Polytechnic Institute.
These two fundamental data types are based on the principle of recursive decomposition[2], and their main fields of usage are: the representation of data used in applications in computer graphics, computer-aided design, robotics and computer vision.

In the following sections, region data (i.e. *2D* shapes) are explained in detail, extending the subject to *3D* data.

3.1 Background

Before discussing some fundamental properties of quadtrees and octrees, we elaborate on the motivation for their development.

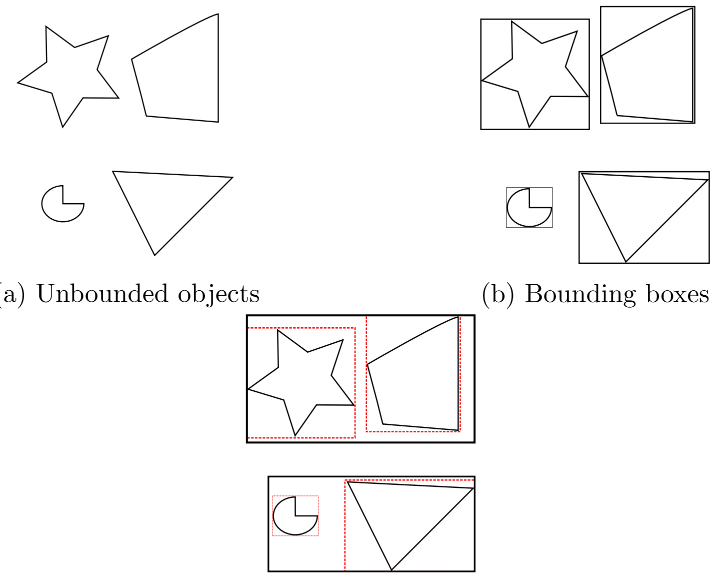
As mentioned earlier, hierarchical data structures such as the quadtree and octree have their roots in attempts to overcome problems that arise when the scene being modeled is more complex than the display grid (in size, precision, number of elements, etc.).
And again, as mentioned earlier, these problems are solved with object-space hierarchies and image-space hierarchies.

- **Object-space hierarchies**. Two kinds of logistic problems present themselves in scene modeling. First, the number of objects to model (i.e, the "universe" of the scene, meaning its totality).
  Secondly, another problem is in determining what subset of the scene is actually visible, further aggravated when the scene extends horizontally and vertically past the bounds of the viewing surface.

  The first problem has been addressed, in part, by observing that the "universe" can be hierarchically organized into objects composed of subobjects, which are in turn composed of other objects, and so forth. This observation has been used as the basis for the organization of the

[1]Described in a 1980 report "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer".

[2]Recursive decomposition is a principle similar to *divide and conquer* methods, and refers to the process whereby, any complex informational event at one level of description, can be specified more fully at a lower level of description by decomposing the event into multiple components and multiple processes, that specifiy the relations among these components.



(a) Unbounded objects (b) Bounding boxes

(c) Hierarchical bounding boxes

Figure 3.1: An example of the use of bounding objects, helpful to solve the visible-subset problem, aggravated by the come out of the scene from the viewing surface. Infact, if the bounding object is not visible, then clearly the object being bounded is also not visible

user's interface of various graphics packages.

And since the object hierarchy must be kept to solve the communication problem between the object and the overall picture to which it belongs, it can also be used to solve the visible-subset problem, adapting the object hierarchy through the notion of bounding objects.

When detemining whether or not an object is visible, it is common to surround the object (see Figure 3.1a) with a bounding box (see Figure 3.1b) or even a sphere. If the bounding object is not visible, then clearly the object being bounded is also not visible.
This technique produces a major computational savings, since it is usually much easier to test for visibility of the bounding object than the visibility of the bounded object.

However, to improve execution time in presence of a great number of elements, the bounds need not be limited to the primitive objects of the scene, but instead, bounding objects can also be placed around complex geometries formed by the different levels of the object hierarchy (see Figure 3.1c).

- **Image-space hierarchies**. A natural alternative to processing in the object-space hierarchy is to organize the data around an image-space hierarchy.
  One problem with traditional image-space representations (i.e., 2D and 3D arrays) is that they require the user to fix the maximum resolution in advance. However, a hierarchical organization of the image space allows the resolution to vary with the complexity of the objects in various regions.

  Of course, there are many ways to partition the image space (viewed as a continuous plane/space), but to interface easily with a Cartesian coordinate system, a decomposition of the plane into square regions (and a space into cubes) is the simplest.

  When justifying the use of object-space hierarchies for image-space processing, we often refer to the property of *area coherence*, which means that objects tend to represent compact regions in the image space.
  Similarly, we might speak of *object coherence* as being a factor in image-space hierarchies, since regions that are close to each other tend to be parts of the same object. Thus, both types of hierarchies tend to approximate each other.

  For large-scale applications, however, the costs associated with the imprecision of these approximations can easily overshadow any benefits accrued from the explicit maintenance of just one of the hierarchies. Thus, when possible, both hierarchies should be maintained.

3.2 Quadtree and Octree Definitions

The term quadtree is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space.

A quadtree is a tree data structure[3] in which each internal node has exactly four children.
Quadtrees are most often used to partition a *2D* space by recursively subdividing it into four quadrants or regions. The data associated with a *leaf cell*[4] varies by application, but the leaf cell represents a "unit of interesting spatial information".

The subdivided regions may be square or rectangular, and the resolution of the decomposition (i.e., the number of times that the decomposition process is applied) may be fixed beforehand or it may be governed by properties of the input data.

Based on the above, quadtree can be differentiated on the following bases:

- the type of data that they represent;
- the principle guiding the decomposition process;
- the resolution (variable or not);
- whether the shape of the tree is independent of the order in which data is processed.

Currently, they are used for point data, regions, curves, surfaces, and volumes. The decomposition may be into equal parts on each level (i.e., *regular polygons*, termed a *regular decomposition*), or it may be governed by the input.

The current section explains the construction of quadtrees, and their most common types.

A **tree-pyramid** (*T-pyramid*) is a "complete" tree. Every node of the T-pyramid has four child nodes except leaf nodes, and all leaves are on the same level. Level, that corresponds to individual pixels in the image.
The data in a tree-pyramid can be stored compactly in an array as an implicit data structure, similar to the way a complete binary tree[5] can be

[3]In computer science, a tree is a widely used abstract data type (ADT) that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

[4]A leaf cell (or *leaf node*) is a term used to indicate any node that does not have child nodes.

[5]A binary tree is a tree with a branching factor of 2, hence binary. In other words, each node can have at most two children.

stored compactly in an array[6].

Consider now an incomplete tree: if we were to store every node corresponding to a subdivided cell, we may end up storing a lot of empty nodes. We can cut down on the size of such sparse trees by only storing subtrees whose leaves have interesting data (i.e. "important subtrees").
But we can actually cut down on the size even further. When we only keep important subtrees, the pruning process may leave long paths in the tree where the intermediate nodes have degree two (a link to one parent and one child). It turns out that we only need to store the node u at the beginning of this path (associating some meta-data with it to represent the removed nodes) and attach the subtree rooted at its end to u.
What just presented is the core idea of **compressed quadtrees**.

The **point quadtree** is an adaptation of a binary tree used to represent *2D* point data.
It is often very efficient in comparing two-dimensional, ordered data points, usually operating in O(log$n$)[7] time, where n is the number of items stored. Point quadtrees are worth mentioning for completeness, but they have been surpassed by *k-d trees*[8] as tools for generalized binary search.
Point quadtrees are constructed as follows. Given the next point to insert, we find the cell in which it lies and add it to the tree. The new point is added such that the cell that contains it is divided into quadrants by the

[6]There is a distinction between a tree as an abstract data type and as a concrete data structure. As a data type, a tree has a value (value of the root) and children, and the children are themselves trees (subtrees of the children of the root node). As a data structure, a tree is a group of nodes, where each node has a value and a list of references to other nodes (its children). Nodes in a tree could have next/previous references, or references to their parent nodes.

[7]Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables

[8]In computer science, a k-d tree (short for *k-dimensional tree*) is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). k-d trees are a special case of binary space partitioning trees.

vertical and horizontal lines that run through the point. Consequently, cells are rectangular but not necessarily square.
In these trees, each node contains one of the input points.

Since the division of the plane is decided by the order of point-insertion, the tree's height is sensitive to and dependent on insertion order. Inserting in a "bad" order can lead to a tree of height linear in the number of input points (at which point it becomes a linked-list). If the point-set is static, pre-processing can be done to create a tree of balanced height.

Region quadtree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on, with each leaf node containing data corresponding to a specific subregion.
Each node in the tree either has exactly four children, or has no children (a leaf node).
The height of quadtrees that follow this decomposition strategy (i.e. subdividing subquadrants as long as there is interesting data in the subquadrant for which more refinement is desired) is sensitive to (and dependent on) the spatial distribution of interesting areas in the space being decomposed.

This data structure is constructed in the following manner: starting with an image (whose binary array representation is given in Figure 3.2a), it is checked to see if it has a simple description and thus does not require any further hierarchical structuring.
If this is not the case, then the image space is partitioned into four disjoint congruent square regions (called quadrants) whose union covers the original image space (see Figure 3.2b).
Each of these new image spaces is treated as if it were isolated, and each one is examined to determine whether or not it has a simple description (resulting in Figure 3.2c). Of course, in this example, the stopping rule for the decomposition process is homogeneity (i.e., each square region is of one color).
This decomposition technique is referred to as a regular decomposition, to distinguish it from decomposition approaches that vary the size of the subregions formed from the original regions (see Figure 3.2d).

The criteria to stop the decomposition process is called *leaf criterion*, because the spaces that satisfy it form the leaf nodes of the tree that represents the hierarchical structure.
There are many variants on the quadtree data structure that differ only in

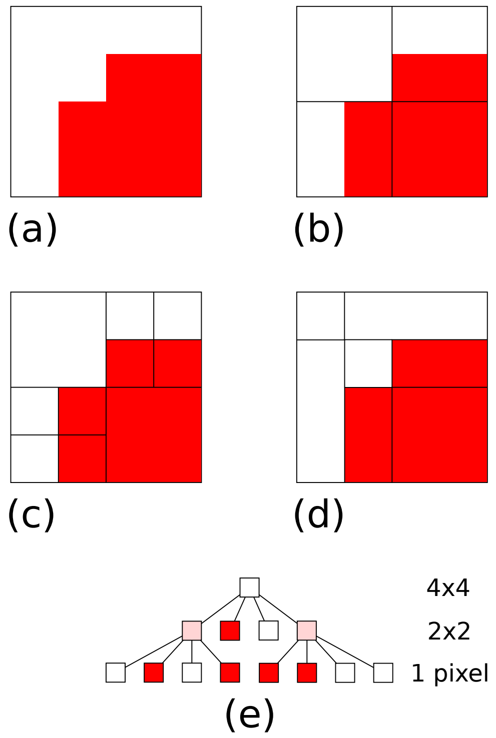


Figure 3.2: Illustration of the quadtree decomposition process: (a) original image, (b) first level of decomposition, (c) second and final level of decomposition, (d) an example of an irregular decomposition, (e) its tree representation

what constitutes a satisfactory leaf criterion for the data structure, being a multitude of plausible (and different) decomposition criteria.

But when looking for a leaf criterion, we should search for a subset of the possible image spaces where the graphics task we want to process can be solved easily, satisfying correctly the starting decomposition criteria. Sometimes, in fact, the leaf criterion cannot be fulfilled in a finite amount of time or resources.

For instance, consider the **edge octree**. This kind of quadtrees are usually used to store lines by subdividing cells to a very fine resolution, specifically until there is a single line segment per cell.
If not set a maximum level of decomposition, near corners/vertices edge

quadtrees will continue dividing until they reach their default leaf criterion: division in square regions where no region contains more than one line segment.
Fixing a maximum level of decomposition, by contrast, make it possible to stop the partitioning, although it can result in extremely unbalanced trees which may defeat the purpose of indexing.
For example, Figure 3.3 is the edge quadtree corresponding to the vector data in red. In this case, truncation at the maximum tree depth (4) has occurred at the nodes containing vertices A, B, C, D, E, F, and G, but not H.

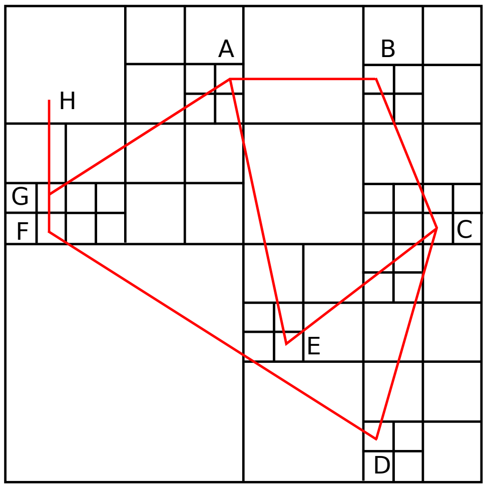


Figure 3.3: The edge quadtree for the vector data in red. The fixed maximum level of decomposition is 4, as can be seen around vertices. If this imposition stops the partitioning, it unfortunately leads to a not so well balanced quadtree

The octree data structure is the 3D analog of the quadtree; its being can easily extrapolated from the concepts previously exposed, and it was developed independently by various researchers.
Hunter[9] mentioned it as a natural extension of the quadtree, and Reddy and Rubin[10] proposed the octree as one of three representations for solid

[9]G.M. Hunter; *Efficient computation and data structures for graphics*, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Prince.ton University, Princeton, NJ, 1978.

[10]D.R. Reddy and S. Rubin, *Representation of three-dimensional objects*, CMU-CS-7S-

objects. The second is a *3D* generalization of the point quadtree of Finkel and Bentley[11], i.e. a decomposition into rectangular parallelepipeds (as opposed to cubes) with planes perpendicular to the x, y, and z axes. The third breaks the object into rectangular parallelepipeds of an arbitrary size that are not necessarily aligned with an axis.
Meagher[12] developed numerous algorithms for performing solid modeling where the octree is the underlying representation.

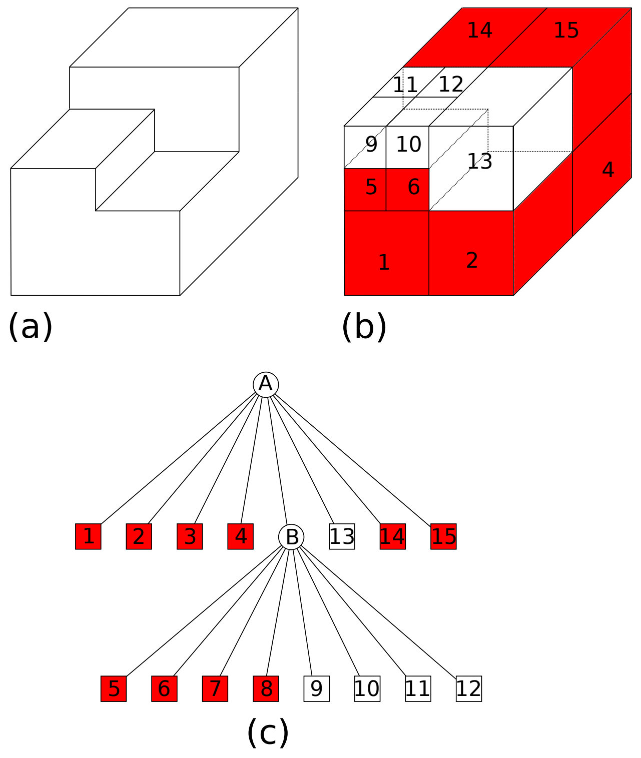


Figure 3.4: (a) Example 3D object, (b) its octree block decomposition, and (c) its tree representation. The original figure (a) is rasterized filling its empy parts with some voxels (represented in white, in (b), while in red are identified the parallelepiped used to divide the original image)

113, Computer Science Department, Carnegie-Mellon University, Pittsburgh, April 1978.

[11]R.A. Finkel and J.L. Bentley, *Quad trees: a data structure for retrieval on composite keys*, Acta Informatica, 1(1974), 1-9.

[12]D . Meagher, Geometric modeling using octree encoding, Computer Draphics and Image Processing 19, 2(June 1982), 129-147.

To be consistent with this chapter, we report an example of costrunction for a region octree.
It starts with an image in the form of a cubical volume, and continues determining if its description is sufficiently complex, in which case the volume is recursively subdivided into eight congruent disjoint cubes (called octants), until the complexity is sufficiently reduced.
Of course, the leaf criteria differ depending on whether the data is of a raster format (consisting of 3D voxels[13] having a single color, instead of 2D pixels) or vector format (consisting of solids and planar or curved surfaces, instead of polygons and edges).
Figure 3.4a is an example of a simple 3D object whose raster octree block decomposition is given in Figure 3.4b and tree representation in Figure 3.4c.

3.3 Quadtree and Octree Space Decomposition

Although a number of different planar decomposition methods exist, quadtree in the form of squares it is used, because it is a planar decomposition that satisfies these two properties:

1. it yields a partition that is an infinitely repetitive pattern, so it can be used for images of any size;
2. it yields a partition that is infinitely decomposable into increasingly finer patterns, so into higher resolution.

A quadtree-like decomposition into four equilateral triangles also satisfies these criteria, and they have been used in the literature (Yamaguchi et al.[14] use them to generate an isometric view from an octree representation of an object).
However, unlike the decomposition into squares, it does not have a uniform orientation[15] (all tiles with the same orientation cannot be mapped into each

[13]In *3D* raster graphics, the volume is divided into evenly spaced rows and columns, covering the three different directions (up-down, left-right, in-out). This divides the 3D space into cubes, also know as voxels (volume elements). Each voxel has a 3D coordinate within the volume and holds the color at that coordinate.

[14]. Yamaguchi, T.L. Kunii, K. Fujimura, and H. Toriya, *Octree-related data structures and algorithms*, IEEE Computer Graphics and Applications, 4, 1(January 1984), 53-59.

[15]Three points p,q,r define a clockwise, a counter-clockwise triangle, or may be aligned. These three cases correspond to a positive, negative, or null value of the following determinant:

$$\begin{matrix} x_q - x_p & x_r - x_p \\ y_q - y_p & y_r - y_p \end{matrix}$$

other by translations of the plane that do not involve rotation or reflection). In contrast, a decomposition into hexagons has a uniform orientation but does not satisfy property 2.

One of the motivations for the development of hierarchical data structures such as the quadtree is a desire to save space (we have already introduced the compressed quadtrees).
Besides consideration of the leaf criteria, the investigation of hierarchical data structures has also been concerned with how to encode the tree representing the hierarchy.
In literature, three general approaches are usually presented to represent trees, each of which has been investigated with regard to the specific representation of quadtrees.
In the following, we describe them just for quadtree, bearing in mind that extension to their *3D* counterpart is trivial.

1. The original formulation of the quadtree encodes it as a tree structure that uses **pointers**. It is the first and most obvious quadtree encoding which requires additional overhead to encode the internal nodes of the tree.
   Each internal node (often referred to as a *gray node*), in fact, requires four pointers (one for each of its subtrees), apart from the leaf nodes, needing no pointer fields.
   The size of a pointer field is the base 2 logarithm of the number of nodes in the tree. Each node also requires one bit of information to indicate whether it is an internal node or a leaf.

   To describe quadtree algorithms, a father link is useful in each node; however, this is not necessary for implementation, because in most tasks, processing starts at the root and a stack of father links can be easily maintained.

2. The second formulation, termed *DF-expression*[16], represents the image in the form of a **traversal of the nodes** of its quadtree.
   Its approach makes use of the observation that the number of subtrees of a given quadtree node, is either four or zero.
   Thus, a quadtree can be represented by listing the nodes encountered

[16]The depth-first picture expression (*DF-expression*, where *DF* stands for *depth first*, or also *treecode*) is a compact quadtree expression providing a high data compression capability. It is primarily a hierarchically structured data representation for binary pictures.

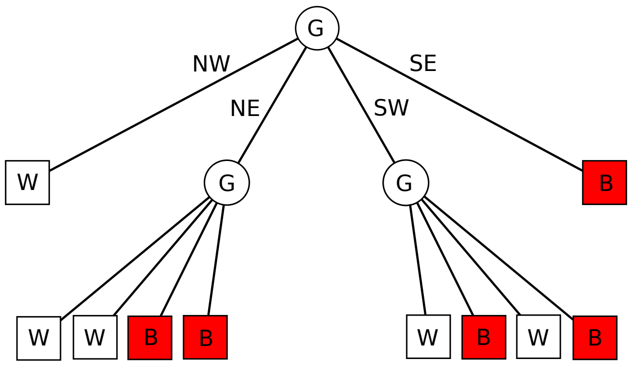


Figure 3.5: Pointer encoding of the quadtree of Figure 3.2c. Internal nodes are represented by circular nodes. Terminal nodes are represented by square nodes

by a preorder traversal[17] of the tree structure.

For example, traversing the quadtree of Figure 3.5 in the order NW, NE, SW and SE, letting G, B and W denote nonterminal, solid and empty nodes, result in the list GWGWWBBGWBWBB.

It is a very compact formulation, as each node type can be encoded with one bit of overhead, used to distinguish between leaf nodes and internal nodes: in the first case, the pre-order algoritm will display the data of the current node; in the second one, it will recursively traverse its left subtree, its right subtree, and finally it will process the current node itself.

Although many simple algorithms are performed and efficiently implemented using this traversal of the quadtree, it is not always easy to use when random access to nodes is desired.
For instance, to visit the second subtree of a node, it is necessary to visit each node of the first subtree so that the location of the root of the second subtree can be determined.

3. The third approach is based on the use of **locational codes**, and it

[17]In computer science, tree traversal (also known as *tree search*) is a form of graph traversal which refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

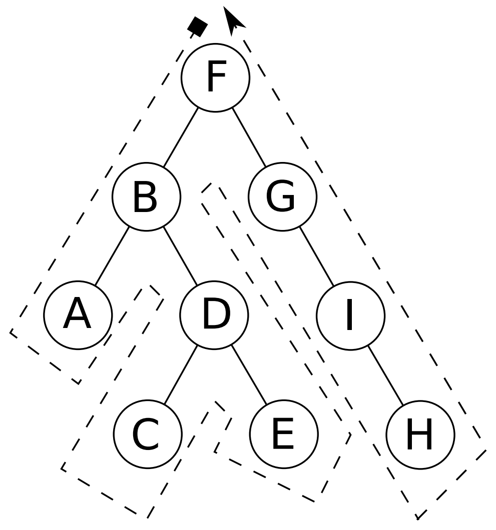


Figure 3.6: Left-to-right (left subtrees are checked before the right ones, as opposed to right-to-left orders) tree traversals. 1) *Pre-order*: F, B, A, D, C, E, G, I, H. 2) *In-order*: A, B, C, D, E, F, G, H, I. 3) *Post-order*: A, C, E, D, B, H, I, G, F

was first proposed by Morton[18] as an index to a geographical database.

Different variant are possibile, but in the one that we describe, each node is represented by a pair of numbers. The first number, termed a locational code, is composed of a concatenation of base 4 digits corresponding to directional codes that locate the node along a path from the root of the quadtree.
These directional codes take on the values 0, 1, 2, and 3 corresponding to quadrants NW, NE, SW, and SE, respectively.
The second number, instead, is the level of the tree at which the node is located.
Assuming that the root is at level 0, the pair of numbers (312,3) are decoded as follows: 312 is the base 4 locational code and denotes a node at level 3 reached by a sequence of transitions SE, NE, and SW- starting at the root.

The overhead per node is two bits per level of depth of the node,

[18]G.M. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*, IBM Ltd., Ottawa, Canada, 1966.

plus the base 2 logarithm of the depth of the node to specify the level at which the node is found. Gargantini[19] has investigated algorithms specific to this representation, which she calls a *linear quadtree*[20][4], because the addresses are keys in a linear list of nodes (in literature, this approach is also call *leafcode*).

This formulation is analogous to taking the binary representation of the x and y coordinates of a designated pixel in the block (e.g., the one at the lower left corner) and interleaving them (i.e., alternating the bits for each coordinate; see Figure 3.8 and Figure 3.9).
Once the data are sorted into this ordering, any one-dimensional data structure can be used, such as binary search trees, B-trees[21], skip lists[22], or hash tables.

In mathematical analysis and computer science, this binary representation is called *Z-order*, *Morton-order*, or *Morton-code*, and is a function which maps multidimensional data to one dimension (as seen for the locational codes), while preserving locality of the data points. The resulting ordering can equivalently be described as the order one would achieve from a depth-first traversal of a quadtree (see Figure 3.6: these searches are referred to as depth-first search (DFS), as the search tree is deepened as much as possible on each child before going to the next sibling).
The Z-order can be used to efficiently build a quadtree for a set of points[7]. The basic idea is to sort the input set according to Z-order and, once sorted, the points can either be stored in a binary search tree and used directly (linear quadtree), or they can be used to build a pointer based quadtree.
As an alternative, the *Hilbert curve* has been suggested, having a better order-preserving behaviour, and giving so a better spatial locality. But the calculations for the Hilbert distance[23] are more complicated

[19]I. Gargantini, *Linear Octtrees for Fast Processing of Three-Dimensional Objects*, Computer Graphics and Image Processing, Dec. 1982, pp. 365-374.

[20]A linear octree is represented by a linear array instead of a tree data structure.

[21]In computer science, a B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children.

[22]In computer science, a skip list is a data structure that allows fast search within an ordered sequence of elements.

[23]Picking a point p covered by the curve, and tracing out the curve until you reach

than for the Morton one, leading to significant processor overhead which, in CPU bound applications, it is not well-liked.
The notion of distance speaking about space-filling curves is important because it can be used as hash from a 2D coordinate to an integer, and because some curves (as said, some more than others) preserve locality, meaning that 2D point nearby in space are likely to have similar Hilbert distances.
For a recent overview on multidimensional data processing, including nearest neighbour searches, refer to [9].

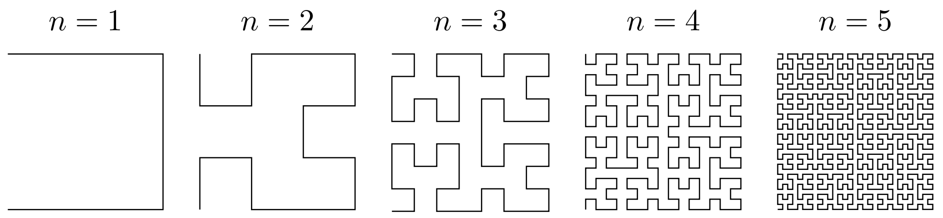


Figure 3.7: Hilbert curve generated using recursive TEX macro and PGF. Author: Marc van Dongen

When using the linear quadtree encoding, further reduction of the storage requirements is possible without substantially increasing the runtime requirements of the algorithms.
In particular, there is no need to retain the internal nodes as the general quadtree structure stores only data in the tree's leaf nodes.
Since the number of internal nodes is equal to one third of the number of leaf nodes minus one, this results in a significant space savings.

Moreover, it is often remarked that the nodes representing a background color (or empty nodes) can also be eliminated from the node list. With a binary raster image, the result is a reduction in the size of the quadtree to one half of its former size (assuming that, on the average, one half of the pixels are background).

The relative compactness of the pointer and the linear quadtree representations depends on the complexity of the scene being represented and on the application in which they are used.

that point, then the number of intermediate points that you passed through, is called the Hilbert distance

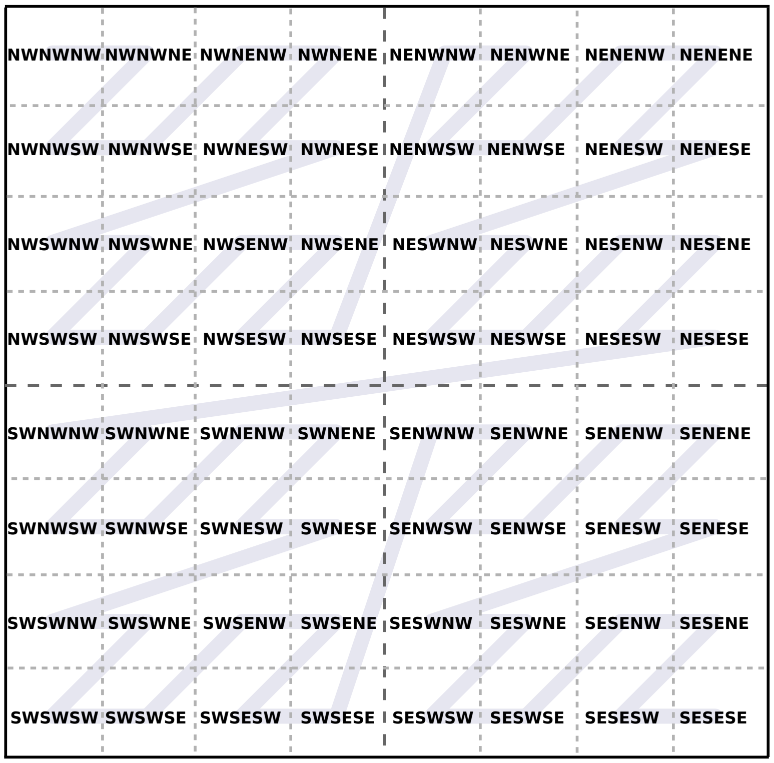
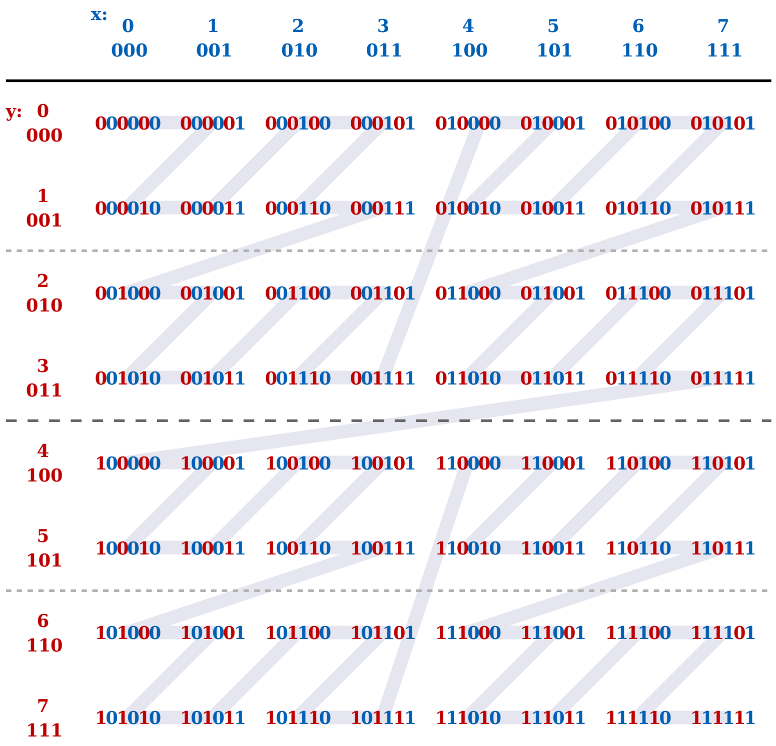

NWNWNW NWNWNE NWNENW NWNENE NENWNW NENWNE NENENW NENENE
NWNWSW NWNWSE NWNESW NWNESE NENWSW NENWSE NENESW NENESE
NWSWNW NWSWNE NWSENW NWSENE NESWNW NESWNE NESENW NESENE
NWSWSW NWSWSE NWSESW NWSESE NESWSW NESWSE NESESW NESESE
SWNWNW SWNWNE SWNENW SWNENE SENWNW SENWNE SENENW SENENE
SWNWSW SWNWSE SWNESW SWNESE SENWSW SENWSE SENESW SENESE
SWSWNW SWSWNE SWSENW SWSENE SESWNW SESWNE SESENW SESENE
SWSWSW SWSWSE SWSESW SWSESE SESWSW SESWSE SESESW SESESE

Figure 3.8: Locational representation for a third level quadtree: NW = 00 (binary 0), NE = 01 (binary 1), SW = 10 (binary 2), SE = 11 (binary 3)

The attractiveness of the linear quadtree representation increases with the complexity of the scene, but the choice is not clear cut, and is further complicated for 3D data, where the overhead of the internal nodes is less of a factor and hence, an efficient implementation of the pointer-based representation will often be more economical spacewise than a locational code representation.

To conclude this section, we present another type of quadtrees, the origin of which is due to the necessity of reducing the number of leaf nodes in these data structures.
The amount of storage required by quadtrees and octrees, infact, is directly proportional to the number of leaf nodes. Reducing the latter, will reduce also the earlier.



y \ x:	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

Figure 3.9: Z-order representation for a third level quadtree (By David Eppstein - Own work, Public Domain). See the correlation with Figure 3.8

And therefore, the **bintree**, rather than splitting a region with respect to all the principal planes simultaneously, it splits a region against only one plane at each level.
For example, instead of splitting an octree node into eight subnodes, the bintree first splits the node into two subnodes along the x-y plane. Each of these subnodes is checked to see if it could be a valid leaf and, if not, is then subdivided along the y-z plane.
Finally, nodes that require further subdivision are subdivided along the x-z plane.
This process is repeated in a cyclical manner until the appropriate maximum level of subdivision is attained.

3.4 The Quadtree/Octree Complexity Theorem

Most quadtree algorithms are simply preorder traversals of the quadtree itself, and thus their execution time is generally a linear function of the number of nodes in the quadtree.
Consequently, we are interested in the asymptotic analysis of the size of a quadtree, more for its relevance to the execution time analysis of the algorithms, than for the amount of storage actually required.
In this case, our discussion assumes a tree representation in the sense that the number of nodes in the quad tree includes the internal nodes.

A key to the analysis of the execution time of quadtree algorithms is the following theorem[13], which states that, except for pathological cases, the number of nodes in the quadtree representation of a region is proportional to the perimeter of the region.

Quadtree Complexity Theorem. *The number of nodes in a quadtree region representation for a simple polygon (i.e. with nonintersecting edges and without holes), is $O(p + q)$ for a $2^q \times 2^q$ image, with perimeter p measured in pixel widths.*

In most cases, q is negligible, and thus, the number of nodes is proportional to the perimeter. Hence, the quadtree complexity theorem holds that the size of the quadtree representation of a region is linear in the perimeter of the region.

An alternative interpretation of this result is that for a given image, if the resolution doubles and hence the perimeter doubles (ignoring fractal effects), then the number of nodes will double. On the other hand, for the *2D* array representation, when the resolution doubles, the size of the array quadruples.
Therefore, asymptotically, quadtrees are arbitrarily more compact than *2D* arrays. Figure 3.10 illustrates the relative growth of the two representations for a simple triangular region.

The Quadtree Complexity Theorem holds also for *3D* data where the perimeter is replaced by surface area, and in general for d-dimensions where, instead of perimeter, we have the size of the (d-1)-dimensional interfaces between the d-dimensional objects.
In particular, most algorithms that execute on a quadtree representation of an image instead of an array representation have an execution time that is

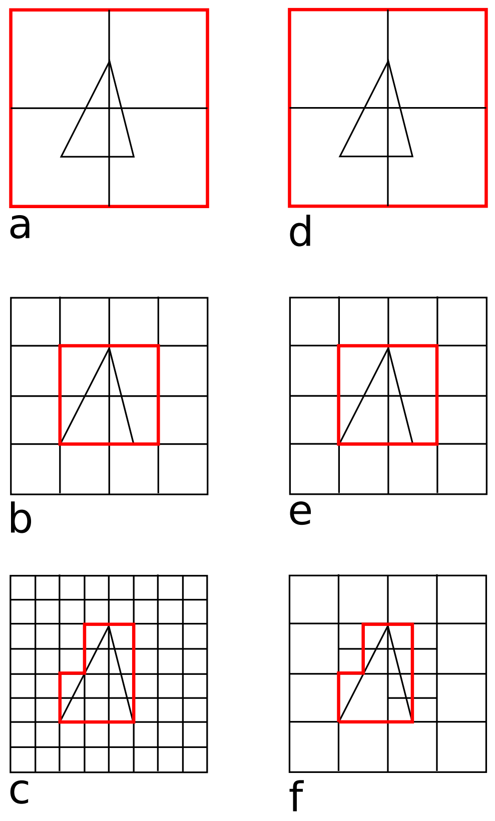


Figure 3.10: An illustration of the relative growth of the array and quadtree representations at different levels of resolution for a simple triangular region. Figures (a) through (c) are the array representations of the triangle at resolutions 1, 2 and 3. Figures (d) through (f) are the corresponding quadtree representations at the same resolutions

proportional to the number of blocks in the image rather than the number of pixels.
Generally, this means that the application of a quadtree algorithm to a problem in d-dimensional space executes in time proportional to the analogous array-based algorithm in the (d-1)-dimensional space of the surface of the original d-dimensional image.

Thus, quadtrees are somewhat like dimension-reducing devices.

3.5 Algorithms Using Quadtrees

In this section we describe how some algorithms can be implemented using quadtrees, with respect to two subjects extremely valuable for meshes generation: point location and neighboring object location.

- **Point location**. Probably, the simplest task to perform on raster data is determining the color of a given pixel.
  In the traditional raster representation, this task is accomplished by exactly one array access, while the quadtree rendering requires searching its internal structure.
  The algorithm starts at the root of the quadtree and uses the values of the x and y coordinates of the center of its block, to determine which of the four subtrees contains the pixel.
  For example, if both the x and y coordinates of the pixel are less than the x and y coordinates of the center of the root's block, then the pixel belongs in the southwest subtree of the root. This process is performed recursively until a leaf is reached, whose color will be the color of the pixel.
  The execution time for the algorithm is proportional to the level of the leaf node containing the desired pixel.

  However, point location can also be performed without the explicit evaluation of the centers of each node encountered along the path. This approach uses the depth n of the pixel we are searching (respect of course to that of the root), and assumes that the southwestern-most pixel is at $(0, 0)$.
  This approach to pixel location is easiest to contemplate with respect to a quadtree representation that makes use of locational codes, although it is equally applicable to the pointer representation of quadtrees.
  The locational code for a leaf is formed by a process (described in the section on data structure implementations) that is equivalent to interleaving the binary coordinates of the lower left-hand corner of the leaf (here coordinates are integer values ranging from 0 to $2^n - 1$ for a $2^n \times 2^n$ grid).
  When the leaf nodes are sorted by their locational codes (as for a preorder traversal of the quadtree), the addresses of all descendants of a node, say P, lie between the address of P and the address of its immediate successor at the same level. Then, a pixel is located by first interleaving the binary representations of its coordinates to construct

an address, say K, for a hypothetical leaf node corresponding to the pixel.
And this hypothetical leaf is located by performing a binary search[24] on the sorted list of locational codes for the leaf nodes of the quadtree and returning the leaf node with the largest locational code value that is less than or equal to K.

The execution time for this algorithm is proportional to the log of the number of leaf nodes in the tree (assuming key comparisons can be made in constant time).
When a pointer representation is used, the execution time is proportional to the level of the leaf node containing the desired pixel[25]. The difference in the execution time is due to the fact that here, the pixel-location algorithm is slightly different. In particular, we locate the appropriate leaf by descending the tree.

- **Neighboring object location**. The vector analog of the pixel-location task is the object-location operation, where the x and y coordinates of the location of a pointing device (e.g., mouse, tablet, light-pen) must be translated into the name of the appropriate object.
To handle this task, we must first determine the leaf that contains the indicated location. If the leaf is empty, then we must investigate other leaf nodes.
Investigation that we have to do even if the leaf node is not empty, unless the location of the pointing device coincides with a primitive[26], because it is possible that a nearer primitive might exist in another leaf.
What we are trying to do, in essence, is wishing to report the nearest primitive of the object description, stored in the quadtree.

The approach used is a top-down[27] recursive algorithm (the opera-

[24]In computer science, binary search, also known as *half-interval search*, *logarithmic search*, or *binary chop*, is a search algorithm that finds the position of a target value within a sorted array. Binary search runs in at worst logarithmic time, making *O(log n)* comparisons, where n is the number of elements in the array.

[25]Finding a node in a linked list is *O(n)* where, in this case, $n = 4^q$, where q is the level at which the desired pixel it is, and 4 is the *2D* repartition of a quadtree.

[26]A primitive point P of a group of point is simply a generator of this group: all elements of the group can be expressed as $P + P + P + ... + P$ (k times), for some k.

[27]Solving a problem by reducing it to one or more simpler problems is the essence of the top-down approach to designing algorithms. One advantage to this approach is that

tion is also known as the *nearest neighbor problem*): initially, at each level of the recursion, we explore the subtree that contains the location of the pointing device, say P. Once the leaf containing P has been found, the distance from P to the nearest primitive in the leaf is calculated (empty leaf nodes have a value of infinity). Next, we unwind the recursion.
As we do so, at each level we search the subtrees that represent regions overlapping a circle centered at P, whose radius is the distance to the closest primitive that has been found so far.
When more than one subtree must be searched, the subtrees representing regions nearer to P are searched before the subtrees farther away (since it is possible that a primitive in them might make it unnecessary to search the subtrees that are farther away).

Consider, for example, Figure 3.11 (it reports only point data; however, the treatment of vector data differs from point data only in the formula used to calculate the distance from a point) and the task of finding the nearest neighbor of P in node 7.

If we visit nodes in the order SW, SE, NW, NE as we unwind for the first time, we check nodes 1 and 2 and the subtrees of the western brother of 7. Once we visit node 3, we have to visit nodes 4, 5 and 6, because they could contain a closer point to P.
By contrast, after have visited node 6, there is no need to visit nodes 8 and 9, since node 6 contained A, which is closer than B.
Same reasoning still applies for node 10, containing point C (which is farther than A), and for nodes 11, 12, 13.

3.6 Quadtree Grid Generations

The algorithms exposed just now, are of great importance in a context of mesh generation to solve partial differential equations by numerical methods.
A well-known field where this resolve process has gained great importance is the *computational fluid dynamics* (*CFD*), where an automatic mesh generation technique which can accommodate local mesh refinement adaptively is desirable, because in many CFD problems it is necessary to fit a mesh of varying cell density to the boundary of curved or irregular domains, in order to obtain accurate numerical solutions to the governing partial differential

it allows to abstract away certain details, to focus on the main steps of the algorithm.

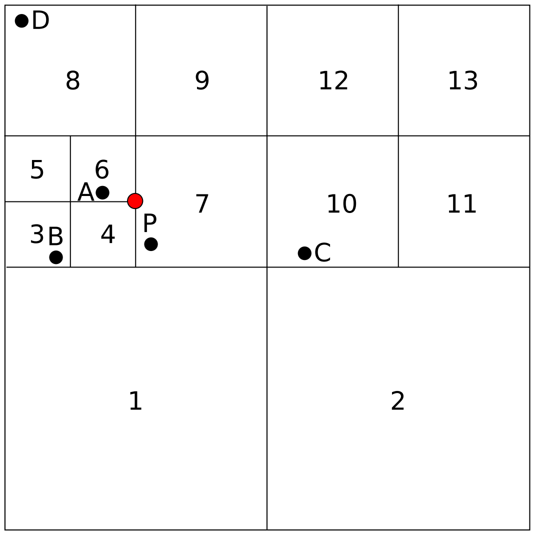


Figure 3.11: Example illustrating the neighboring object problem: P is the location of the pointing device, for which we want to find the closest neighbor in the surrounding area. To succeed in our quest, we can use a recursive algorithm to easily find the nearest object, which is represented by point A in node 6

equations without requiring an excessive number of mesh points.
But not only that: regions with a rapid change in the flow variables are not known in advance, and they are identified during the solution process; thus, the mesh should be adapted dynamically until the solution is sufficiently accurate.
And of course, the mesh must be quick to generate and easy to adapt. Therefore, an automatic mesh generation technique which can keep track of the refined cells and control the local level of refinement is essential. Such a technique must establish links between neighbouring cells at different levels in a hierarchical sense to facilitate local mesh refinement.

One efficient technique for producing such meshes in two-dimensional space is precisely to subdivide recursively the domain into quadrants using a quadtree to store and manipulate the mesh information.

By the way, the transition from an algorithmic and theoretical point of view about quadtrees (such as it has been addressed so far in this chapter), to a more pragmatic and practic approach like this, is not for free.

Indeed, there are two main difficulties in using this grid generation technique:

- quadtrees are complicated in terms of storing and retrieving mesh information due to their recursive tree structures.
- Hanging nodes[28] (see the red point in Figure 3.11) arise at the interface between cells of different sizes in quadtree grids.

To overcome these issues, CFD developers can handle software libraries which take in charge of all the challenges concerning the domain discretization for their study cases.
As far as quadtrees (and octrees) are concerned, there are a lot of tools which can be used, but in continuing this thesis, just one will be taken into account: **PABLO** (and of course, its Python counterpart **PABLitO**).

PABLO is C++/MPI library for parallel linear octree/quadtree developed by Optimad Engineering srl under the GNU Lesser General Public License, and it is now contained in **bitpit**[29]. It's strengths are:

- to provide users with a ready-to-use tool for parallel adaptive grid of quadrilaterals/hexahedra;
- message passing paradigm is transparent to the user since PABLO has embedded MPI calls. By this way, the user can easily perform data communications and dynamic load-balance by calling straightforward high level methods;
- the user can feel free to customize his data in whatever way he likes;
- to allow adaptive mesh refinement by generating non-conforming grid with hanging nodes;
- low memory consumption in the basic configuration (approx. 30B per octant in 3D).

[28]Hanging nodes are vertices of the smaller cells which lie on the face of an adjacent larger cell but not on any of its vertices. These nodes cause problems when discretizing partial differential equations.

[29]Bitpit is a C++ library for scientific High Performance Computing. Within bitpit, different modules factorize the typical effort which is needed to derived a real-life application code. Main efforts are dedicated to handle differnt types of computational meshes, their runtime adaptation and data transfer for parallel applications.

- 2:1 balancing[30] between octants and a easy way to generate and store intersections between octants;
- the existence of **PABLitO**[31], that is a Python wrapper for PABLO, written in Cython and developed as main framework for the work of this thesis.

PABLitO is, as far as I know, the first parallel linear octree which can be easily imported in Python. There is Pyoctree[32], but this is an octree structure containing a 3D triangular mesh model, to be used for ray tracing[33].
Like the already mentioned "rival", PABLitO requires a C++ compiler and Cython, to be operational, since both are centered around a C++ implementation, to improve computational speed.
But, unlike Pyoctree, which can take opportunity from an OpenMP support, PABLitO uses a distributed approach (if an MPI implementation is present on the system, otherwise it will run in serial), leaving PABLO managing the intra-communications inside each octree, and bearing the inter-communications between octrees (see Figure 4.11). And this is a real strength, because until now I have never heard about multiple communicators in an MPI programs, leaving to the classic COMM_WORLD all the communications' management.
In the next subsection it will be presented a typical starting point for the usage of this new Python module

3.6.1 PABLitO Initialization

Creating differents MPI intra-communicators

```
group_w = comm_w.Get_group()
procs_w = comm_w.Get_size()
procs_w_list = range(0, procs_w)
procs_l_lists = chunk_list_ordered(procs_w_list,
                                   n_grids)
proc_grid = get_proc_grid(procs_l_lists,
                          comm_w.Get_rank())
group_l = group_w.Incl(procs_l_lists[proc_grid])
```

[30]An octree is 2:1 balanced if there are no pair of adjacent blocks where one is more than double the size of the other.

[31]Written by Federico Tesser: https://github.com/uncleTes.

[32]Written by Michael Hogg: https://github.com/mhogg/pyoctree.

[33]In computer graphics, ray tracing is a rendering technique for generating an image by tracing the path of light as pixels in an image plane and simulating the effects of its encounters with virtual objects.

```
# Creating differents MPI intracommunicators.
comm_l = comm_w.Create(group_l)
# Current intracommunicator's name.
comm_name = comm_names[proc_grid]
comm_l.Set_name(comm_name)
#Communicator local's name.
comm_l_n = comm_l.Get_name()
#Communicator local's rank.
comm_l_r = comm_l.Get_rank()
#Communicator global's name.
comm_w_n = comm_w.Get_name()
```

where

- *procs_l_lists* is a list containing the processes' lists associated to each octree
- *proc_grid* is the integer prepresenting between the corresponding octree of the current MPI process
- *group_l* is the MPI processes group local to each octree
- *comm_l* is the local intra-communicator build on the previously built local group

Creating different MPI inter-communicators

```
# Creating differents MPI intercommunicators.
intercomm_dictionary = {}

if procs_w > 1:
    create_intercomms(n_grids      ,
                      proc_grid    ,
                      comm_l       ,
                      procs_l_lists,
                      logger       ,
                      intercomm_dictionary)
```

where

- *intercomm_dictionary* is the Python dictionary containig all the inter-communicators at stake. The key of each pair is a unique and safe tag (created specifically by an intrinsic algorithm)
- *create_intercomms* is the function to populate *intercomm_dictionary*

Setting PABLOs

```
pablo, centers = set_octree(comm_l,
                            proc_grid)

...
...
...
```

where

- *set_octree* is the method to set a PABLO for each process

3.7 Summary

This chapter has introduced quadtrees (and octrees) as data structures originated from images representation fields, examining their data representation and space decomposition, and subsequently setting out different areas of applications, including point and object location, and grid generations. Of this latter, which is the most important aspect for the continuation of the thesis, a grounding in the libraries used in the following chapters has been given, showing also some specific code for the usage of the carrier framework of this project.

Bibliography

[1] Carsten Burstedde. Forest-of-octrees amr: algorithms and interfaces, 2012. Second HPC Workshop KAUST.

[2] Carsten Burstedde. Modular forest-of-octrees amr: algorithms and interfaces, 2012. FEniCS, Simula Research Laboratory, Norway.

[3] Omar Ghattas Carsten Burstedde, Lucas C. Wilcox. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 2011.

[4] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 1982.

[5] Robert E. Webber Hanan Samet. Hierarchical data strucutres and algorithms for computer graphics. *IEEE Computer Graphics & Applications*, 1988.

[6] S. Cruz A. Saalehi A. G. L. Borthwick K. F. C. YJU, D.M. Greaves. Quadtree grid generation: Information handling, boundary fitting and cfd applications. *Computers & Fluids*, 1996.

[7] D. Eppstein M. Bern. Parallel construction of quadtrees and quality triangulations. *Int. J. Comp. Geom. & Appl*, 1999.

[8] Hari Sundar Ilya Lashuk George Biros Rahul S. Sampath, Santi S. Adavani. Dendro: Parallel algorithms for multigrid and amr methods on 2:1 balanced octrees. *SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[9] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan-Kaufmann, 2006.

[10] Hanan Samet. An overview of quadtrees, octrees, and related hierarchical data structures. *Theoretical Foundations of Computer Graphics and CAD*, 1988.

[11] Lucas C. Wilcox Omar Ghattas Tobin Isaac, Carsten Burstedde. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 2015.

[12] Omar Ghattas Tobin Isaac, Carsten Burstedde. Low-cost parallel algorithms for 2:1 octree balance. *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, 2012.

[13] Paul E. Black Vreda Pieterse. Quadtree complexity theorem. *Algorithms and Theory of Computation Handbook*, 2004.

Chapter 4

Overlapping Octrees

BSC	Barcelona Supercomputing Center
KAUST	King Abdullah University of Science and Technology
LTE	Leading Truncation Error
ASCII	American Standard Code for Information Interchange
VTK	The Visualization Toolkit
SIMD	Single Instruction Multiple Data
MIMD	Multiple Instruction Multiple Data
FTCS	Forward Time Centered Space
BTCS	Backward Time Centered Space
UDS	Upwind Difference Scheme
CDS	Central Difference Scheme

In this chapter we will cover in detail the new adaptive method which is the key issue of this thesis.

Method that aims to reduce 1) the complexity typical of the Chimera aprroaches (where the overall algorithm can be not so easy to be implemented), and 2) the increasing number of the in place computational grids (which, as we have seen in Section 1.3, can be a complexity hot spot for the communication pattern between all the meshes).

And of course this method seeks to mark a little turning point in the *CFD* programming techinques, leaving opened a glimmer on the usage of non-"classical" (of course, relatively speaking) programming languages, having been completely developed using Python and Cython.
These two languages were used with the aim of presenting a real scientific computing application totally operating, as already told, with a language (Python) suitable for easy developing, easy training, and user-friendliness. All aspects that, longer than I, have affected the way of thinking of large companies like NVIDIA and Intel, and of important research groups and scientists from, for example, BSC, Argonne National Laboratory and KAUST.

Although in different fileds, all the already mentioned interested subjects are unified by the desire to create Python's wrappers, to use their original products, having been dazzled by its versatility and power.

And so, why not to try to emulate them, in a real contest application?

After having introduced the main points of modeling and simulation of physical processes, and after having established the key points for the numerical simulation of partial differential equations (PDEs) and for the *finite difference* and *finite volume* numerical methods, we will move into the technicalities and the detailed aspects of this 2D discretization method based on the overlap of generic quadrilateral grids made by quadtrees which, as we already seen in Chapter 1, can be compared to block structured *AMR*, but that in reality differs from it in the usage of the intrinsic capability of the quadtrees themselves to be refined locally, using for example some error estimator or some prediction functions, or maybe just some shape functions.

4.1 PDEs Models and Simulations

Every physical process that we want to resolve must be, at first, ordered onto the following stages:

- the **definition** of the physical problem, listing its governing equations, to which follows then
- the **creation** of the mathematical model using systems of *PDEs* or

ODEs, each of which combined with initial and (or) boundary conditions (to get a well-posed problem[1]).
PDEs are, in fact, mathematical models of continuous physical phenomenon in which a dependent variable, say u, is a function of more than one independent variable (and of their derivatives), say t (time), and x, y, z (spatial position).
A special case is ordinary differential equations (ODEs), which deal with functions of a single variable.

- After the previous steps, before solving the underlying problem, it is necessary to **discretize** the mathematical model, generating a discrete grid on which to project the continuous scheme of the governing equations, giving rise to a sistem of algebraic equations.
  This step is necessary when there are no other ways, other than the numerical one, to **solve** the physical process (complex problems usually do not have analytical solution). And all the numerical techniques for PDEs are based on the creation of a discrete space alongside the continuous one in which the governing equations originally belong.

- Finally, after having obtained the keenly awaited solution, it remains to **analyse** the obtained results, looking for a comparison with the predicted ones or, if presents, with the analytical reference solution.

PDEs can be used to describe a wide variety of phenomena such as sound, heat, electrostatics, electrodynamics, fluid dynamics, elasticity, or quantum mechanics. These seemingly distinct physical phenomena can be formalised similarly in terms of PDEs, giving rise to mainly three families of them.
Given the following second order[2] general form for linear PDEs in two variables

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} + d\frac{\partial u}{\partial x} + e\frac{\partial u}{\partial y} + fu + g = 0$$

we define the **discriminant** $b^2 - 4ac$, on the basis of which we can classify them into the following groups.

- **Hyperbolic**: $b^2 - 4ac > 0$.

[1]The mathematical term well-posed problem stems from a definition given by Jacques Hadamard, and it means that mathematical models of physical phenomena should underlie the properties that 1) a solution exists, 2) the solution is unique and 3) the solution's behavior changes continuously with the initial conditions.

[2]The order of a partial differential equation is the order of the highest derivative occurring in the equation.

Hyperbolic PDEs describe time-dependent, conservative physical processes, such as convection[3], that are not evolving toward steady state.

An example is the one dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = 0$$

- **Parabolic**: $b^2 - 4ac = 0$.

  Parabolic PDEs describe time-dependent dissipative physical processes, such as diffusion[4], that are evolving toward steady state.

  An example is the one dimensional heat equation

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0$$

- **Elliptic**: $b^2 - 4ac < 0$.

  Elliptic PDEs describe processes that have alreay reached steady states, and hence are time-independent.

  An example is the two dimensional Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

In the following section we will introduce two numerical techniques for solving PDEs: *finite difference* and *finite volume* methods.

[3]in a general convection-diffusion equation, the contribution of convection is described by $-\nabla \cdot (\bar{v}u)$. Roughly speaking, to understand it imagine standing on the bank of a river, measuring the amount of salt of the water each second. If upriver somebody dumps a bucket of salt into the river, later you would see the salinity suddenly rise, then fall, as the zone of salty water passes by. Thus, the concentration at a given location can change because of the flow.

[4]Diffusion, by the way, is usually represented by $\nabla \cdot (\nabla u)$. If we imagine u as the concentration of a chemical, when it is low somewhere compared to the surrounding areas (a local minimum of concentration), the substance will diffuse in from the surroundings (that are evolving toward steady state), so the concentration will increased. And viceversa, if concentration is higher than the surroundings.

4.2 Numerical Techniques for PDEs

Numerical partial differential equations is the branch of numerical analysis that studies the numerical solution of partial differential equations which, as we have already seen, not necessarily present an analytical settlement, and therefore it is necessary to obtain an approximation of the behaviour of the desired solution.

Of course different approaches are possible, but here we will focus only on two of them, very similar in the underlying resolving approach.

4.2.1 Finite Difference Method

Finite difference method represents functions by their values at certain grid points and derivatives are approximated through differences in these values.

Suppose that we are solving $u = u(t, x)$ on the domain $\Omega = [0, T] \times [0, L]$. We discretize the domain Ω by partitioning the spatial interval $[0, L]$ into $m + 2$ grid points $x_0, x_1, \ldots, x_m, x_{m+1} = L$, such that

$$\Delta x_j = x_{j+1} - xj, \quad j = 0, 1, 2, \ldots m$$

In the case that the $m + 2$ spatial points x_j are equally spaced, we have $\Delta x = \Delta x_j, \forall j$ (see Figure 4.1).

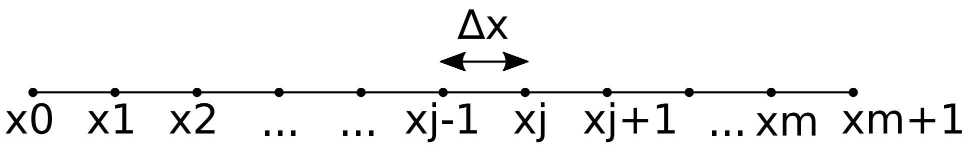


Figure 4.1: Finite difference discretization for the one dimensional domain $[0, L]$. Similarly is discretized the temporal domain $[0, T]$, with time step $k = \Delta t$.

The numerical solution of the PDE is an approximation to the exact solution obtained using a discrete representation at the grid points x_j of the discrete spatial mesh, at every time level t_k.
Let us denote this numerical solution as U, such that

$$U_j^k \approx u(t_k, x_j)$$

For each time level t_n, so, the numerical solution is a collection of **finite** values:

$$U^n = [U_1^n, U_2^n, \ldots, U_m^n]$$

collection in which, values U_0^n and U_{m+1}^n are given by boundary conditions, for all the time steps.
On the other and, the initial condition, at time 0, U^0, must be given for all the spatial grid points.

Recalling that the derivative of a function u at the point x_j is defined by the limits

$$\begin{aligned} u_x(x_j) &= \lim_{h\to 0} \frac{u(x_j+h)-u(x_j)}{h} \\ &= \lim_{h\to 0} \frac{u(x_j)-u(x_j-h)}{h} \\ &= \lim_{h\to 0} \frac{u(x_j+h)-u(x_j-h)}{2h} \end{aligned}$$

we use them, with a small finite value of $h = \Delta x$, to define the three different approximation schemes of the finite difference method (a geometric clarification is reported in figure 4.3):

$$\begin{aligned} u_x(x_j) &\approx \frac{u(x_j+\Delta x)-u(x_j)}{\Delta x} && (\textbf{forward difference}) \\ &\approx \frac{u(x_j)-u(x_j-\Delta x)}{\Delta x} && (\textbf{backward difference}) \\ &\approx \frac{u(x_j+\Delta x)-u(x_j-\Delta x)}{2\Delta x} && (\textbf{centered difference}) \end{aligned}$$

whose characteristic is the error obtained approximating the derivative.
The **leading truncation error** (**LTE**) is in fact obtained using *Taylor* approximation[5] around a grid point x_j. So, considering for the moment forward and backward approximation, we have:

$$u_{j+1} = u_j + \Delta x \left(\frac{\partial u}{\partial x}\right)_j + \frac{(\Delta x)^2}{2}\left(\frac{\partial^2 u}{\partial x^2}\right)_j + \frac{(\Delta x)^3}{6}\left(\frac{\partial^3 u}{\partial x^3}\right)_j + \cdots \qquad (1)$$

$$u_{j-1} = u_j - \Delta x \left(\frac{\partial u}{\partial x}\right)_j + \frac{(\Delta x)^2}{2}\left(\frac{\partial^2 u}{\partial x^2}\right)_j - \frac{(\Delta x)^3}{6}\left(\frac{\partial^3 u}{\partial x^3}\right)_j + \cdots \qquad (2)$$

[5] $u(x) = \sum_{n=0}^{\infty} \frac{(x-x_j)^n}{n!}\left(\frac{\partial^n u}{\partial x^n}\right)_j$.

from which we get the accuracy of these two finite difference approximations, rearranging the previous equations in the following way

$$\left(\frac{\partial u}{\partial x}\right)_j = \frac{u_{j+1}-u_j}{\Delta x} - \frac{\Delta x}{2}\left(\frac{\partial^2 u}{\partial x^2}\right)_j - \frac{\Delta x^2}{6}\left(\frac{\partial^3 u}{\partial x^3}\right)_j + \cdots$$

$$\left(\frac{\partial u}{\partial x}\right)_j = \frac{u_j-u_{j-1}}{\Delta x} + \frac{\Delta x}{2}\left(\frac{\partial^2 u}{\partial x^2}\right)_j - \frac{\Delta x^2}{6}\left(\frac{\partial^3 u}{\partial x^3}\right)_j + \cdots$$

where in red is represented the leading truncation error, expressed in the form:

$$\epsilon_\tau = \alpha_m (\Delta x)^m + \alpha_{m+1} (\Delta x)^{m+1} + \cdots \approx \alpha_m (\Delta x)^m$$

It is so clear then that for the forward and backward finite difference methods, the truncation error is $O(\Delta x)$, meaning that they are first order approximations.
For the central difference scheme, an estimation of its LTE is obtained subtracting the previous two formulae, (1) − (2), obtaining

$$\left(\frac{\partial u}{\partial x}\right)_j = \frac{u_{j+1}-u_{j-1}}{2\Delta x} - \frac{\Delta x^2}{3}\left(\frac{\partial^3 u}{\partial x^3}\right)_j + \cdots$$

which clearly reveals its nature more precise ($O\left(\Delta x^2\right)$).

Summarising what has been done previously, it must be clear that:

- the backward and forward difference are a first order accurate approximation to the partial derivative u_x at x_j, and that their LTEs are $O(\Delta x)$;
- the centered difference is a second order accurate approximation to the partial derivative u_x at x_j, and that its LTE is $O\left(\Delta x^2\right)$.

If on the other hand, we add (1) and (2), rather than subtract them, we obtain a central approximation for the second order derivatives

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_j = \frac{u_{j+1}-2uj+u_{j-1}}{\Delta x^2} + O\left(\Delta x^2\right)$$

which can be used, for example, considering the heat equation $u_t = u_{xx}$, where we can apply a forward difference at time t_n and a central one in space,

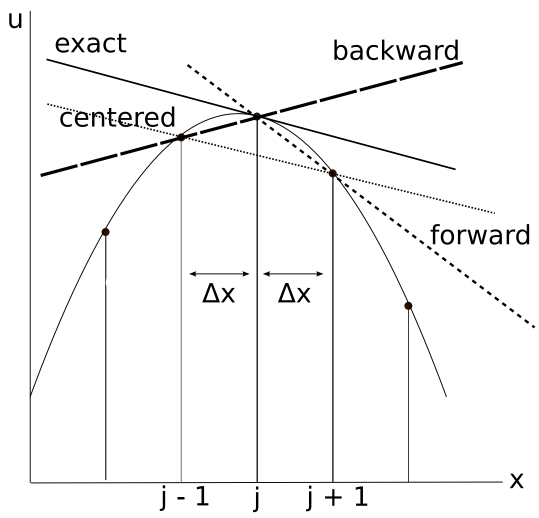


Figure 4.2: Geometric clarification for the three finite difference approximations: derivative at point *j* can be evaluated using the information from the previous point *j - 1* (backward), from the following one *j + 1* (formward, or from both of them (centered). Of course, reducing the distances between evaluation points, more accurate approximations are obtained

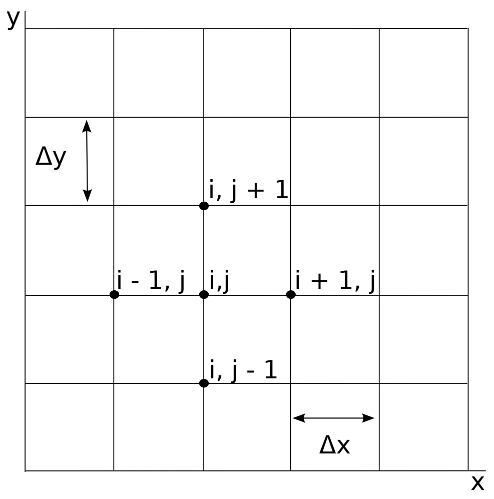


Figure 4.3: 2D equations are discretized easily extending the concepts introduced for the 1D finite difference method, on a 2D domain decomposition: the x derivative evaluated on j, will be $u_x\left(x_{i,j}\right) = \dfrac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}$

to obtain the following **explicit** (solution at time level $k+1$ is determined by solution at previous time levels only) scheme:

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} = \frac{u_{j+1}^k - u_j^k + u_{j-1}^k}{\Delta x^2} \rightarrow$$

$$u_j^{k+1} = u_j^k + \frac{\Delta t}{\Delta x^2}\left(u_{j+1}^k - u_j^k + u_{j-1}^k\right)$$

We can note that this scheme is first order accurate in time and second order accurate in space.
Both the time and space derivatives are replaced by finite differences. Doing so requires to specify both the time and spatial locations of the u values in the finite difference formulas. Therefore, we have introduced a superscript k to designate the time step of the discrete solution.
The previous *Forward Time, Centered Space*(**FTCS**) formulation is easy to implement because the values of u_j^{k+1} can be updated indipendently of each other, not depending on u_{j-1}^{k+1} or u_{j+1}^{k+1}.
If this is a good thing from a programming point of view, for the stability of the solution this is not. In fact, explicit approximation require the following condition to be satisfied, in order to obtain stable solutions.

$$\frac{\Delta t}{\Delta x^2} < \frac{1}{2}$$

To involve bot the current state and the later one to find the solution at time $k+1$ (therefore, an **implicit** scheme is introduced), we could use a backward difference at time t_{n+1}, remaining fixed in space:

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} = \frac{u_{j+1}^{k+1} - u_j^{k+1} + u_{j-1}^{k+1}}{\Delta x^2}$$

This approach is imaginatively called *Backward Time, Centered Space* (BTCS), and it requires to solve a system of equations at each time step. It goes without saying that this means a computational effort greater than the one in the FTCS, and also the code developing will be more complicated.
However, even if this scheme is as just as accurate as the previous one, it is unconditional stable, more robust then to the choice of Δt and Δx (for solutions of the heat equation).

4.2.2 Finite volume method

Finite volume method represents and evaluates partial differential equations in the form of algebraic equations, by acting in a similar fashion as in the finite difference method, except for the fact that here, rather than pointwise approximations on a grid, we have the average integral value on a **reference** (or **control**) **volume**. The method starts by dividng the flow domain into

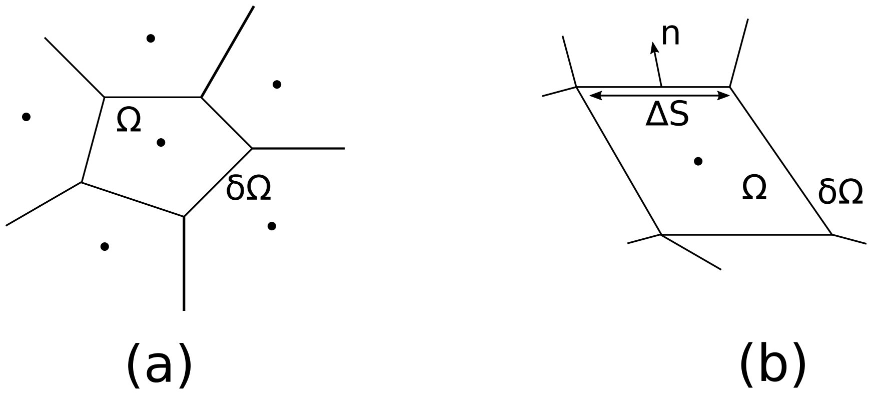


Figure 4.4: Control volumes Ω, boundaries $\partial\Omega$, normals n to the edges, having surfaces ΔS, are the ingredients distinctive of a finite volume resolution

a number of small control volumes, in whose centers are defined the grid points where, usually, unknown are stored.
The equations are then integrated over each control volume. Considering for example a general (steady) transport equation[6] for some quantity ϕ of the type $\nabla \cdot \left(\bar{U}\phi\right) = \nabla \cdot (\gamma \nabla \phi) + S_\phi$, if we integrate we obtain

$$\int_\Omega \nabla \cdot \left(\bar{U}\phi\right) dV = \int_\Omega \nabla \cdot (\gamma \nabla \phi)\, dV + \int_\Omega S_\phi dV \quad \rightarrow$$

$$\int_\Omega \nabla \cdot \left(\bar{U}\phi - \gamma \nabla \phi\right) dV = \int_\Omega S_\phi dV$$

where the last passage is achieved combining together the convection $(\nabla \cdot \left(\bar{U}\phi\right))$ and difussion $(\nabla \cdot (\gamma \nabla \phi))$ terms.

The *divergence theorem*[7] can be used to convert the left hand side volume

[6]The convection–diffusion equation is a combination of the diffusion and convection (advection) equations, and describes physical phenomena where physical quantities are transferred inside a physical system due to two processes: diffusion and convection.

[7]In vector calculus, the divergence theorem (also known as Gauss's theorem), is a result that relates the flow of a vector field through a surface, to the behavior of the vector field inside that surface. Precisely, the divergence theorem states that the outward flux of a vector field, through a closed surface, is equal to the volume integral of the divergence over the region inside the surface.

integral to an integral around the boundary $\partial\Omega$

$$\int_{\partial\Omega} \left(\bar{U}\phi - \gamma\nabla\phi\right) \cdot \bar{n} dS = \int_{\Omega} S_\phi dV \tag{4.1}$$

which is a statement of conservation for the control volume, being $\bar{U}\phi \cdot \bar{n}dS$ and $\gamma\nabla\phi \cdot \bar{n}dS$ respectively the convective and diffusive flux across the same boundary part, and being their integral the total net fluxes into the control volume.
Provided that the same expressions are used for fluxes across faces in neighbouring cells, this approach ensures that the conservation properties will also be satisfied globally over the flow domain. In the Equation 4.1 we have

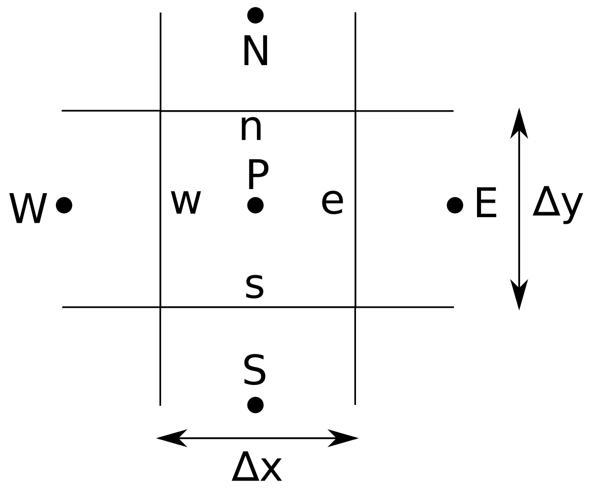


Figure 4.5: Finite volume compass notation: neighbours of P are indicated with the cardinal directions N, W, S, E, while cell faces are denoted with the corresponding lower cases

that:

- the surface integral is generally approximated in a discretized form by

$$\int_{\partial\Omega} \left(\bar{U}\phi - \gamma\nabla\phi\right) \cdot \bar{n} dS \approx \sum_k \left(\bar{U}\phi - \gamma\nabla\phi\right)_k \cdot \left(\bar{n}\Delta S\right)_k$$

  where the flux $\left(\bar{U}\phi - \gamma\nabla\phi\right)_k$ is evaluated at the centre of the edge k of the volume Ω, and $(\Delta S)_k$ and $\bar{n}_k$ are, respectively, the area and the unit vector normal to the edge k (referring to Figure 4.5, $k = e$, n, w, s).

But now, an issue occurs: if the solution is available only at computational nodes (the centers of the control volumes), we need an interpolation to obtain the convection fluxes at the evaluation points, namely the centers of the edges of the control volumes.
To do this, different approximations are present in literature, and we will show two of them, of the first and second order, to keep consistency with the finite differences section.

The first one is the *upwind difference scheme*(**UDS**); considering a typical grid point j in a one-dimensional domain (see Figure 4.6), there are only two directions associated with point j: left (towards negative infinity) and right (towards positive infinity).
If velocity $\bar{U}$ is positive, then the left side is called upwind side and the right side is the downwind side. Viceversa if u is negative.
And speaking about the approximations done, the following ones are taken

$$\begin{aligned}
&\text{if } \bar{U} > 0 \quad \rightarrow \qquad \phi_{j-1/2} \approx \phi_{j-1} \quad \text{and} \quad \phi_{j+1/2} \approx \phi_j \\
&\text{if } \bar{U} < 0 \quad \rightarrow \qquad \phi_{j-1/2} \approx \phi_{j} \quad \text{and} \quad \phi_{j+1/2} \approx \phi_{j+1}
\end{aligned}$$

which are first-order accurate flux approximations, as the leading truncation error shows in the following Taylor expansion

$$\bar{U}\phi_{j+1/2} = \bar{U}\phi_j - \frac{\bar{U}\Delta x}{2}\left(\frac{\partial \phi}{\partial x}\right)_{j+1/2} - \bar{U}\frac{(\Delta x)^2}{8}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_{j+1/2} + \cdots$$

The second one is the *central difference scheme*(**CDS**) which reduces a classical polynomial interpolation $p_1(x) = \phi_L \frac{x_R - x}{x_R - x_L} + \phi_R \frac{x - x_L}{x_R - x_L}$ to the averaged values

$$\phi_{j-1/2} \approx \frac{\phi_{j-1} + \phi_j}{2}$$

$$\phi_{j+1/2} \approx \frac{\phi_{j1} + \phi_{j+1}}{2}$$

from where, using the following Taylor expansions

$$\phi_{j+1} = \phi_{j+1/2} + \frac{\Delta x}{2}\left(\frac{\partial \phi}{\partial x}\right)_{j+1/2} + \frac{(\Delta x)^2}{8}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_{j+1/2} + \cdots$$

$$\phi_{j} = \phi_{j+1/2} - \frac{\Delta x}{2}\left(\frac{\partial \phi}{\partial x}\right)_{j+1/2} + \frac{(\Delta x)^2}{8}\left(\frac{\partial^2 u\phi}{\partial x^2}\right)_{j+1/2} + \cdots$$

we obtain the second order LTE

$$\phi_{j+1/2} = \frac{\phi_j + \phi_{j+1}}{2} - \frac{(\Delta x)^2}{8}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_{j+1/2} + \cdots$$

adding the previous two equations.

- The source term is approximated as

$$\int_\Omega S_\phi dV \approx \bar{S_\phi} Vol \approx (S_\phi)_P \, Vol$$

where $\bar{S_\phi}$ is the average value of S_ϕ over the control volume, and $(S_\phi)_P$ is simply its value ath the cell centre node P.

This approximation can also be shown to be usually of the second order.

As an example, we can consider the elliptic equation $u_{xx} = f(x)$ on the control volume $V_i = [x_{i-1/2}, x_{i+1/2}]$. Applying the revious steps, we will have:

$$\int_{x_{i-1/2}}^{i+1/2} u_{xx} dx = \int_{x_{i-1/2}}^{i+1/2} f dx \quad \rightarrow$$

$$u_x(x_{i+1/2}) - u_x(x_{i-1/2}) = (x_{i+1/2} - x_{i-1/2}) f_i \quad \rightarrow$$

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h} = h f_i$$

where the last transition is obtained using centered differences on the derivatives, and the *midpoint rule*[8].

[8] $\int_a^b f(x)\,dx \approx (b-a)\, f\left(\frac{a+b}{2}\right)$.

To conclude, finite difference methods are relatively easy to apply, when the flow geometry allows a simple Cartesian set of grid points, to be adopted. But, if complex geometries are present, more work is needed, not being able to arrange orthogonal grid lines to easily approximate derivatives. Therefore, other solutions must be found.

And of course, in engineering problems, special care is given to ensure conservation properties.
Finite volumes, being based on applying conservation principles over each control volume, guarantee global conservation, but not only. As each reference volume can be assigned different material parameters, they are the natural choice for heterogeneous material.

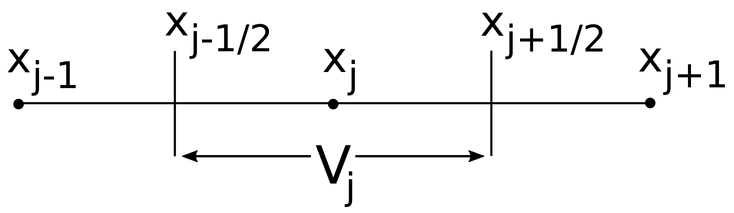


Figure 4.6: 1D control volume V_j with its borders $x_{j-1/2}$ and $x_{j+1/2}$. x_j is its center, while x_{j-1} and x_{j+1} are the center of the closer volumes

4.3 Poisson Equation

In this section we start introducing the main mathematical topic of this work: the **Poisson equation**.

Many books on programming languages starts with an "Hello world" program, so it seems normal to start a new numerical PDEs resolution approach from one of the most fundamental equations: the Poisson one, indeed.
It will be our counterpart to the classical "first program ever", that programmers love.

In mathematics, Poisson equation is a partial differential equation of elliptic type with broad utility in mechanical engineering and theoretical physics.
The Poisson equation arises in numerous physical contexts, including heat conduction, electrostatics, diffusion of substances, twisting of elastic rods, inviscid fluid flow, and water waves.

It is a generalization of Laplace's equation (previously seen), which is also very frequently seen in physics, and it is named after the French mathematician, geometer, and physicist Siméon Denis Poisson.
Although it is very common and very well studied, this equation keeps a certin importance appearing in numerical splitting strategies for more complicated systems of PDEs, in particular the *Navier-Stokes* equations.

Its mathematical formulation is the following:

$$\Delta u\left(\bar{x}\right) = f\left(\bar{x}\right) \quad x\, in\, \Omega$$

$$u\left(\bar{x}\right) = u_d\left(\bar{x}\right) \quad x\, on\, \partial\Omega$$

where $u = u\left(\bar{x}\right)$ is the unknown function, $f = f\left(\bar{x}\right)$ is a forcing term, Ω is the spatial domain and $\partial\Omega$ is its boundary.

The Poisson equation, made up by both the PDE $-\Delta u = f$ and the boundary condition $u = u_d$, is an example of *boundary-value problem* which, in the field of differential equations, is a differential equation together with a set of additional constraints, called the boundary conditions.
A solution to a boundary value problem is a solution to the differential equation which also satisfies the boundary conditions, which can be of three different types:

- **Dirichlet**: also called a first-type boundary condition, it specifies the value of the function itself. Assuming an unknown function y, we could have

$$y = f$$

- **Neumann**: this a second-type boundary condition, and it specifies the value of the normal derivative of the function, like

$$\frac{\partial y}{\partial n} = f$$

- **Robin**: a mix of the previous two so, supposed c_0 and c_1 some constants, whe could have

$$c_0 y + c_1 \frac{\partial y}{\partial n} = f$$

In two space dimensions with coordinates x and y, we can write the Poisson equation as

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f\left(x, y\right) \tag{4.2}$$

where the unknown $u = u\left(x, y\right)$ is a function of two variables and it is defined over a two-dimensional domain Ω.

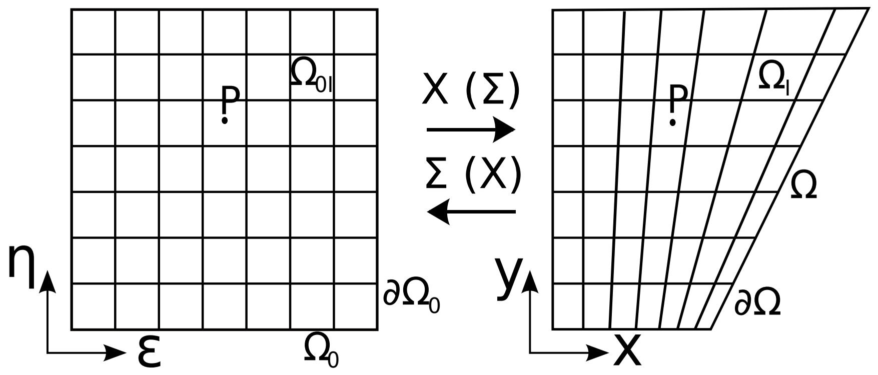


Figure 4.7: Computational Ω_0 and physical Ω domains, where $\partial\Omega_0$ and $\partial\Omega$ represent their corresponding boundaries. Ω_{0i} and Ω_i are, instead, computational and physical control cells. A direct (physical to computational) and an inverse (computational to physical) mapping between the domains are mentioned by $\Sigma\left(X\right)$ and $X\left(\Sigma\right)$

4.4 Numerical Discretization of the Poisson Equation

Equation 4.2, with the corresponding Dirichlet boundary condition, specifically in 2D space, is the pivot around which we have developed the job done so far, following the two different numerical approaches we have previously introduced:

1. finite-difference approach:

$$\frac{\partial^2 u}{\partial \epsilon^2}\left[\left(\frac{\partial \epsilon}{\partial x}\right)^2+\left(\frac{\partial \epsilon}{\partial y}\right)^2\right]+\frac{\partial^2 u}{\partial \eta^2}\left[\left(\frac{\partial \eta}{\partial x}\right)^2+\left(\frac{\partial \eta}{\partial y}\right)^2\right]+$$

$$2\frac{\partial^2 u}{\partial \eta \partial \epsilon}\left[\frac{\partial \eta}{\partial x}\frac{\partial \epsilon}{\partial x}+\frac{\partial \eta}{\partial y}\frac{\partial \epsilon}{\partial y}\right]+\frac{\partial u}{\partial \epsilon}\left[\frac{\partial^2 \epsilon}{\partial^2 x}+\frac{\partial^2 \epsilon}{\partial^2 y}\right]+$$

$$\frac{\partial u}{\partial \eta}\left[\frac{\partial^2 \eta}{\partial^2 x}+\frac{\partial^2 \eta}{\partial^2 y}\right]=f$$

and

2. finite-volume approach:

$$\int_{\Omega_i} \Delta u = \int_{\Omega_i} f$$

$$\int_{\partial \Omega_{0i}} \nabla_\Sigma u \left(\nabla_\Sigma X\right)^{-1} \mathrm{C}\left(\nabla_\Sigma X\right) n_0 \mathrm{d}s_0 = \int_{\Omega_{0i}} f(X(\Sigma))|\nabla_\Sigma X|$$

$$\sum_{j=1}^{4}\left[\nabla_\Sigma u \left(\nabla_\Sigma X\right)_j^{-1} \mathrm{C}\left(\nabla_\Sigma X\right)_j\right] n_{0j} s_{0j} = \left[f(X(\Sigma))|\nabla_\Sigma X|\right]_p \Omega_{0i}$$

where we already have considered, in the discretization, the grid transformations (see Figure 4.7) which take into account a direct (physical to computational, $\Sigma(X)$,) and an inverse (computational to physical, $X(\Sigma)$) mapping between the domains.
This procedure is necessary because derivatives on a cartesian grid $(\frac{\partial u}{\partial \epsilon}, \frac{\partial u}{\partial \eta})$ are easy to compute, while those on a body fitted mesh $(\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y})$ are more difficult to be evaluted, but they are more concrete, reflecting real cases.
So, it would be convenient to re-write the PDE of the case in terms of ϵ and η, instead of x and y, and to discretize it in the computational domain rather than in the physical one (taking into account, of course, derivative transformations).

4.4.1 From/To Computational To/From Physical Domains

Given therefore the mapping

$$\epsilon = \epsilon(x, y) \qquad \eta = \eta(x, y)$$

and the chain rule

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial \epsilon}\frac{\partial \epsilon}{\partial x} + \frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial x} \qquad \frac{\partial u}{\partial y} = \frac{\partial u}{\partial \epsilon}\frac{\partial \epsilon}{\partial y} + \frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial y} \tag{4.3}$$

we have the following second order derivatives transformations

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial \epsilon}\frac{\partial^2 \epsilon}{\partial x^2} + \frac{\partial u}{\partial \eta}\frac{\partial^2 \eta}{\partial x^2} + 2\frac{\partial^2 u}{\partial \epsilon \partial \eta}\frac{\partial \epsilon}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial^2 u}{\partial \epsilon^2}\left(\frac{\partial \epsilon}{\partial x}\right)^2 + \frac{\partial^2 u}{\partial \eta^2}\left(\frac{\partial \eta}{\partial x}\right)^2$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{\partial u}{\partial \epsilon}\frac{\partial^2 \epsilon}{\partial y^2} + \frac{\partial u}{\partial \eta}\frac{\partial^2 \eta}{\partial y^2} + 2\frac{\partial^2 u}{\partial \epsilon \partial \eta}\frac{\partial \epsilon}{\partial x}\frac{\partial \eta}{\partial y} + \frac{\partial^2 u}{\partial \epsilon^2}\left(\frac{\partial \epsilon}{\partial y}\right)^2 + \frac{\partial^2 u}{\partial \eta^2}\left(\frac{\partial \eta}{\partial y}\right)^2$$

which explain the previous finite difference (re)formulation. The matrix

$$\begin{bmatrix} \frac{\partial \epsilon}{\partial x} & \frac{\partial \epsilon}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix}$$

is the so called *Jacobian matrix* $\nabla_X \Sigma$, which is the matrix of all first-order partial derivatives of a vector-valued function.

Regarding by contrast the finite volume re-elaboration, the change of variable $X(\Sigma)$, makes it possibile to come back at the reference configuration, considering the determinant of the Jacobian $|\nabla_\Sigma X|$.
However, the following part of the earlier equation

$$\int_{\Omega_i} f = \int_{\Omega_{0i}} f(X(\Sigma))|\nabla_\Sigma X|$$

is valid just for volumes. For surfaces, the general formulation for a change of variable is given by

$$\int_{\partial\Omega_i} f = \int_{\partial\Omega_{0i}} f(X(\Sigma))C\left(\nabla_\Sigma X\right)$$

where $C(\nabla_\Sigma X)$ is the *Cofactor matrix.*
A cofactor matrix of a square matrix is another square matrix whose generic element at position i, j is the **cofactor** associated at the same position of the starting matrix, thus defined:

$$cof_{i,j} = (-1)^{i+j}\, det\left(A_{i,j}\right)$$

where we have called the initial matrix A, to facilitate the explanation. The cofactor matrix is therefore

$$C\left(A\right)=\begin{bmatrix} cof_{1,1}\left(A\right) & cof_{1,2}\left(A\right) & \cdots & cof_{1,n}\left(A\right) \\ \vdots & \ddots & \vdots & \vdots \\ cof_{n,1}\left(A\right) & cof_{n,2}\left(A\right) & \cdots & cof_{n,n}\left(A\right) \end{bmatrix}$$

which, if A is reversible, can be obtained by its *inverse matrix* $adj\left(A\right)$, following these equivalences:

$$adj\left(A\right)=\left(C\left(A\right)\right)^T$$

$$A^{-1}=det\left(A\right)^{-1}adj\left(A\right)$$

$$\boxed{C\left(A\right)=det\left(A\right)\left(A^{-1}\right)^T}$$

where $adj\left(A\right)$ is the *adjoint matrix*[9]. So, considering the Jacobian $\nabla_\Sigma X$ in 2D, we have that

$$\left(\nabla_\Sigma X\right)^{-1}=\frac{1}{det\left(\nabla_\Sigma X\right)}\begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial x}{\partial \eta} \\ -\frac{\partial y}{\partial \epsilon} & \frac{\partial x}{\partial \epsilon} \end{bmatrix} \rightarrow$$

$$C\left(\nabla_\Sigma X\right)=\begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \epsilon} \\ -\frac{\partial x}{\partial \eta} & -\frac{\partial x}{\partial \epsilon} \end{bmatrix}$$

which gives a better concrete representation of the elements used in the finite-volume approach discretization.

To conclude this section, we have to figure out the last term that comes into play considering the finite-volume change of variable $\left(\nabla_\Sigma X\right)^{-1}$. Recalling the chain rule 4.3, we can write it as

$$\begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix}=\begin{bmatrix} \frac{\partial}{\partial \epsilon} \\ \frac{\partial}{\partial \eta} \end{bmatrix}\nabla_X\Sigma$$

[9]In linear algebra, the *adjugate*, *classical adjoint*, or *adjunct* of a square matrix is the transpose of its cofactor matrix.

and repeating it around the other way, we will have that

$$\begin{bmatrix} \dfrac{\partial}{\partial \epsilon} \\ \dfrac{\partial}{\partial \eta} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{bmatrix} \nabla_\Sigma X$$

from which follows that

$$\begin{bmatrix} \dfrac{\partial}{\partial \epsilon} \\ \dfrac{\partial}{\partial \eta} \end{bmatrix} = \left(\begin{bmatrix} \dfrac{\partial}{\partial \epsilon} \\ \dfrac{\partial}{\partial \eta} \end{bmatrix} \nabla_X \Sigma \right) \nabla_\Sigma X$$

so

$$(\nabla_X \Sigma)\ (\nabla_\Sigma X) = I \quad \rightarrow \quad \nabla_X \Sigma = (\nabla_\Sigma X)^{-1}$$

and therefore $(\nabla_\Sigma X)^{-1}$ is nothing but the right Jacobian for the transformation considered.

After having introduced the numerical approaches that have been followed for the problem at the center of our study, in the following section the programming and algorithmic overtures adopted will be expressed, in order to originate something "new" in the wide variety of resolvent methods for the Poisson equation.
Before that, nevertheless, a brief introductory section on the ways present in Python to communicate objects between processes (and especially on the why to use them) is reported.
This will be useful to better explain the underlying parallel scheme, used in the program.

4.5 Communicating Python Objects

Jobs running in different processes have their own independent memory spaces, and in general cannot share data through memory.
To make processes communicate, programmers need some sort of channel. One possible channel would be a shared memory segment, but it's more common to use *serialization*[10], to let the processes "speak" between them, partly bacause, talking about from a programmer point of view, it is usually easier to use and apply.

[10]Serialization is the process of converting an object to another representation (often binary, but it can be expressed using other forms like xml.

4.5.1 Pickle and Marshalling

The Python standard library supports different mechanisms for data persistence[11], many of which rely on disk storage. But **pickling** and **marhsaling**, however, can also work with memory buffers.

- The `pickle` module implements binary protocols for serializing and de-serializing a Python object structure. "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, while "unpickling" is the inverse operation, in which a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.
  Pickling and unpickling are alternatively known as *serialization*, *marshalling*, or*flattening*, and they provide user-extensible facilities to serialize general Python objects using ASCII or binary formats, ensuring to be backwards compatible across Python releases.

- The `marshal` module is another primitive Python serialization module, existing primarily to support Python's *.pyc* files. It provides facilities to serialize built-in Python objects using a binary format specific to Python, independent of machine architecture issues, but its serialization format is not guaranteed to be portable across Python versions.
  Being in fact its primary job to support *.pyc* files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise.

Users should prefer the usage of the `pickle` module, because it keeps track of the objects it has already serialized (so that later references to the same object won't be serialized again), having implications on both recursive objects and object sharing.

- Recursive objects are objects that contain references to themselves, and these are not handled by `marshal`, and even attempting to marshal a recursive object will crash your Python interpreter.

- Object sharing instead, happens when there are multiple references to the same object in different places in the object hierarchy being serialized.
  `pickle` stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which

[11]Serialization is not persistence, but persistence is one way it can be used.

can be very important for mutable objects[12].
Moreover, `marshal` cannot be used to serialize user-defined classes and their instances, while `pickle` can save and restore class instances transparently.

4.5.2 Multiprocessing

Different multiprocesses paradigms exists in Python, which use `pickle` to communicate objects.

One choice could be using the `multiprocessing` module, that is a package that supports spawning processes using an API similar to the threading module, and it runs on both Unix and Windows.
It offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock (remember it?) by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine.
Hereafter an example to communicate between processes using a `Queue`, to pass messages back and forth (ny pickleable object can pass through a Queue). To be exact, in this simple example one message is passed to one worker, then the main process wait for it to finish.

```

import multiprocessing

class MyFancyClass(object):

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print 'Doing something fancy in %s for %s!' %
                                        (proc_name, self.name)


def worker(q):
    obj = q.get()
    obj.do_something()


```

[12]Not all python objects handle changes the same way. Some objects are mutable, meaning they can be altered; others are immutable, meaning that they cannot be changed but rather they return new objects when attempting to update.

```python
if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

However, in accordance with its portability, with its widespread use, and with its common layer shared with all the HPC applications, in this framework it has been decided to adopt (as yet mentioned) an MPI approach, using its binding for Python given by MPI4Py.

4.5.3 MPI4Py

MPI for Python can communicate any built-in or user-defined Python object taking advantage of the features provided by the `pickle module`. These facilities will be routinely used to build binary representations of objects to communicate (at sending processes), and restoring them back (at receiving processes).

Although simple and general, the serialization approach (i.e., pickling and unpickling) previously discussed imposes important overheads in memory as well as processor usage, especially in the scenario of objects with large memory footprints[13] being communicated.
Pickling general Python objects, ranging from primitive or container built-in types to user-defined classes, necessarily requires computer resources. Processing is also needed for dispatching the appropriate serialization method (that depends on the type of the object) and doing the actual packing.

[13]The word footprint generally refers to the extent of physical dimensions that an object occupies, giving a sense of its size. In computing, the memory footprint of a software application indicates its runtime memory requirements, while the program executes. This includes all sorts of active memory regions like code segment containing (mostly) program instructions (and occasionally constants), data segment (both initialized and uninitialized), heap memory, call stack, plus memory required to hold any additional data structures (such as symbol tables, debugging data structures, open files, shared libraries mapped to the current process, and so on) that the program ever needs while executing and will be loaded at least once during the entire run.

Additional memory is always needed, and if its total amount is not known a priori, many reallocations can occur. Indeed, in the case of large numeric arrays, this is certainly unacceptable and precludes communication of objects occupying half or more of the available memory resources.

MPI for Python supports direct communication of any object exporting the single-segment buffer interface[14].
This interface is a standard Python mechanism provided by some types (e.g., strings and numeric arrays), allowing access in the C side to a contiguous memory buffer (i.e., address and length) containing the relevant data. This feature, in conjunction with the capability of constructing user-defined MPI datatypes describing complicated memory layouts, enables the implementation of many algorithms involving multidimensional numeric arrays (e.g., finite difference schemes) directly in Python, with negligible overhead, and almost as fast as compiled Fortran, C, or C++ codes.

Here a simple example, with `pickle` under the hood to serialize the `data` list, followed by a similar one written in C++

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
data = None
send = comm.send
recv = comm.recv
if (rank == 0):
    data = range(0, 10)
    send(data, dest = 1,
         tag = 0)
elif (rank == 1):
    print("Before recv:")
    print(data)
    data = recv(source = 0,
    tag = 0)
    print("After recv:")
    print(data)
```

```cpp
#include "mpi.h"

#include <stdio.h>

int main(int argc, char **argv) {
```

[14]Single segment buffer interface is an obscure name to mean a piece of memory that is contiguous.

```
    int rank, size;
    int x[10]  = {};
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        for (int i=0; i<10 ; i++) x[i] = i;
        MPI_Send(x, 10, MPI_INT, 1, 0,
        MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        printf("Before recv:")
        for(int loop = 0; loop < 10; loop++)
            printf("%d ", x[loop]);
        MPI_Recv(x, 10, MPI_INT, 0, 0,
        print("After recv:")
        for(int loop = 0; loop < 10; loop++)
            printf("%d ", x[loop]);
        MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

and whose output results

```
$ mpirun -n 2 python mpi4py_02.py

        Before recv:
           None
         After recv:
        [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

It has to be noted the ease in the usage of MPI functions, compared with the C++ version (comparison that applies to MPI initialization and finalization too): an easy import of the module `mpi4py` enable the usage of all the classes, subclasses, functions and submodules present in the bindings written by Lisandro Dalcin.

However, The previous MPI4Py example does not take into account contiguous memory buffer data types, while we have seen that they should be used to reduce annoying overheads.
So, here the counterpart using precisely Numpy arrays

```
from mpi4py import MPI
import numpy
```

```python
from numpy import arange, float64
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
send = comm.Send
recv = comm.Recv
if (rank == 0):
    data = arange(5, dtype = float64)
    send([data, 3, MPI.DOUBLE], dest = 1,
         tag = 0)
elif (rank == 1):
    data = 10 * arange(5, dtype = float64)
    print("Before Recv:")
    print(data)
    recv(data, source = 0, tag = 0)
    print("After Recv:")
    print(data)
```

whose output is

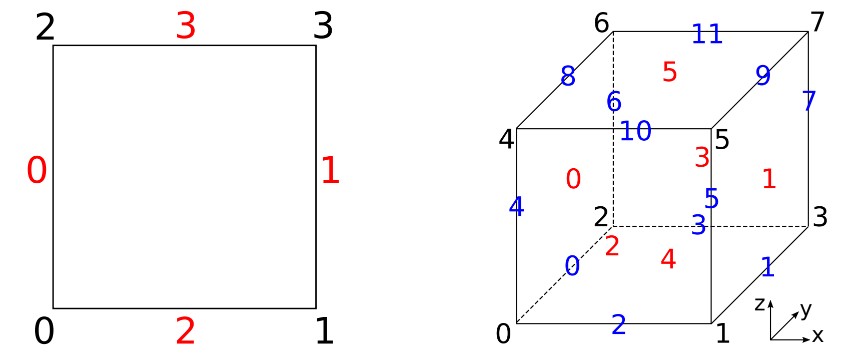
```
$ mpirun -n 2 python mpi4py_03.py

        Before Recv:
   [ 0.  10.  20.  30.  40.]
         After Recv:
           [ 0.  1.  2.  30.  40.]
```

Of course the simplicity of the examples presented it is just for clarification, and not for practical applications.
Actually, in a real scenario, it is desirable to communicate, as well as arrays or contiguous memory data, also structures or objects, without breaking them into individual arrays (even if, this option, for vector optmisation, should be taken into account, but this is totally another topic).
In the following section we will discuss this subject in a more detailed why, explaining why users should do this, and above all how they could do this.

4.6 Algorithmic and Programming Approaches

The starting point of this section is to consider the computational domain of reference as a combination of overlapping quadtrees, whose shapes and positions are decided by the user.

The end user in fact have at his disposal an input file, in which he can set some parameters, relevant for the resolving of the problem.
Below is reported a screenshot of this auxiliary file.



(a) Bitpit 2D cell enumeration (b) Bitpit 3D cell enumeration

Figure 4.8: Numbering example for a 2D cell of a quadtree, and for a 3D volume of an octree. In black is reported the ordering of the nodes, while in red the sorting of the faces, and in blue the numbering of the edges (faces and edges are enumerated following an x-y-z order)

```
[PABLO]
NumberOfGrids =
GridPoints =
Refinements =
[PROBLEM]
Dimension =
Log =
```

It can be seen that there are two dinstinct main sections, referring respectively to the data needed by the quadtrees (`[PABLO]`), and to the data dedicated to the more general "ensemble" of the problem (`[PROBLEM]`).
Entry `NumberOfGrids` is the total number of quadtrees currently involved of which, if n represents their total number, 1 is for the background, and $n-1$ are for the more refined, superposed foreground grids.
`GridPoints` is a set of ternary groups of floating points, representing the four (or eight, in 3D) edge nodes (starting form the anchor node in the lower left corner, see Figure 4.8). The field `Refinements`, by the way, represents a list of integers (one for each quadtree), indicating the starting level of decomposition (see Figure 3.2) of the grids.
Changing section, we have `Dimension` and `Log` (and, for time dependent problems, there will be also the times's informations like step, starting and ending interval) that serve to choose the dimensional environment (2D or 3D), and to enable (or not) the logging on all the parts of the program.

4.6.1 Programming Approach

After having filled up the required fields, the modules containing all the bindings necessary for the simulation, written and compiled in Cython (this last passage is done using `distutils`, a module part of the standard library and useful to build Python packages, and the `cythonize` compiler), are easily imported in the Python interpreter, as a classic one, and allow the access to all the routines and classes, defined in them.
Specifically, two binding modules have been writen, respectively for PABLO and for another library written toujours by Optimad Engineering, helpful with the parallel writing of the *.vtk* files[15], vital for the post-processing and visualization of the results.

Hereafter the body of the function used to build up "custom Cython" extensions from the input *.pyx* files. Here, module `Cython.Distutils` is used instead of the previously mentioned `distutils` and `cythonize`, but the gist does not change.
This function can be used thanks to the "altered" version of the class `build_ext` belonging right to the module used.
Changes to this class have been made specifically for the context of this concerned project.

```
# Define \"Extension\" being cythonized.
def def_ext_modules(self):

    os.environ["CXX"] = "c++"
    os.environ["CC"] = "gcc"
    BITPIT_ENABLE_MPI = 0
    include_libs = "-I" + self.PABLO_include_path
    include_libs = include_libs + " -I" + self.IO_include_path

    if ((not (not self.mpi_include_path)) and \
        (ENABLE_MPI4PY)):
        BITPIT_ENABLE_MPI = 1
        include_libs = include_libs + " -I" + \
                       self.mpi_include_path
        os.environ["CXX"] = "mpic++"
        os.environ["CC"] = "mpicc"
```

[15]The Visualization Toolkit provides a number of source and writer objects to read and write popular data file formats, as well as some of its own file formats. There are two different styles of file formats available in VTK; the simplest are the legacy, serial formats that are easy to read and write either by hand or programmatically. Besides, the XML formats are preferred because they support random access, parallel I/O, and portable data compression.

```

    _extra_compile_args = ["-std=c++11",
                           "-O3"       ,
                           "-fPIC"     ,
                           include_libs,
                           "-DBITPIT_ENABLE_MPI=" + \
                                str(BITPIT_ENABLE_MPI)]
    _extra_link_args = ["-fPIC"]
    _cython_directives = {"boundscheck": False,
                          "wraparound": False,
                          "nonecheck": False}
    _language = "c++"
    _extra_objects = ["libbitpit_MPI.a" if (BITPIT_ENABLE_MPI)
                      else "libbitpit.a"]

    src_dir = os.path.dirname(self.IO_include_path.rstrip("/"))
    common_dir =  src_dir + "/common/"
    operators_dir = src_dir + "/operators/"
    containers_dir = src_dir + "/containers/"
    _include_dirs=["."                   ,
                   self.PABLO_include_path,
                   self.IO_include_path   ,
                   common_dir             ,
                   operators_dir          ,
                   containers_dir         ,
                   numpy_get_include()    ]

    _cc_time_env = {"BITPIT_ENABLE_MPI": BITPIT_ENABLE_MPI}

    splitext = os.path.splitext
    ext_modules=[Extension(splitext(self.extensions_source)[0],
                 [self.extensions_source]                     ,
                 extra_compile_args = _extra_compile_args     ,
                 extra_link_args = _extra_link_args           ,
                 cython_directives = _cython_directives       ,
                 language = _language                         ,
                 extra_objects = _extra_objects               ,
                 include_dirs = _include_dirs                 ,
                 cython_compile_time_env = _cc_time_env       ,
                )]

    return ext_modules
```

while here a piece of code from a Cython “para_tree.pyx” file, just to understand a little better how exactly the writing of a Python wrapper takes place, on the basis of the existence of a C++ library.

```
cdef extern from "ParaTree.hpp" namespace "bitpit":
    cdef cppclass ParaTree:
        ...
        ...
        void computeConnectivity()

cdef class Py_Para_Tree:
    cdef ParaTree* thisptr
    cdef MPI_Comm mpi_comm
    ...
    ...
    def compute_connectivity(self):
        self.thisptr.computeConnectivity()
```

Class `Py_Para_Tree` will be available to the user just writing the following line into the Python interpreter (or in a Python script)

```
from para_tree import Py_Para_Tree
```

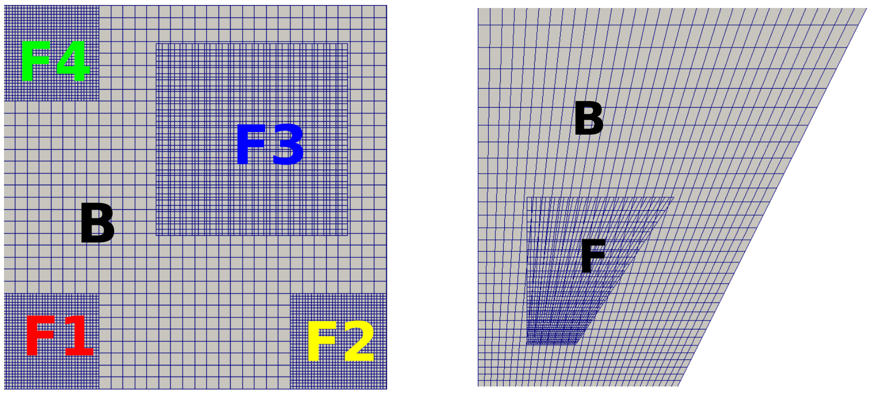
that is, as we said, as easy as the loading of any other Python module.

4.6.2 Algorithmic Approach

After having clarified more specifically the programming modus operandi and efforts, let's focus on the algorithmic part.
As previously said, the computational domain of reference is seen as a combination of overlapping quadtrees, each of which "live" without knowing nothing but its own space and boundaries. Nothing is never said about the existence of other quadtrees.
Thus, for example, in Figure 4.9a any of the defined grids does not know to be in a system made of 5 different physical spaces.
And the problem to be resolved, in our case the Poisson problem, will be equally share between the quadtrees involved. Speaking in a parallel computing fashion, we could equate this approach with the **SIMD** approach. Single Instruction Multiple Data (SIMD), means in fact that all parallel units share the same instruction, but they carry it out on different data elements. Here then, the parallel units would be the quadtrees, while the instruction would be represented by the Poisson problem.
But internally, each instance of the Poisson problem on a different reference domain, in turn will be managed by different processes, recalling the **MIMD** architecture: Multiple Instruction Multiple Data (MIMD) means that parallel units have separate instructions, so each of them can do something different. Indeed, each process will work on a different sub-domain of the target geometry, being able to do something different from the other processes, working on the other sub-domains.

To achieve this in parallel, MPI *inter* and *intra* communicators are used. Intra-communicators will handle communications internal at each quadtrees, while inter-communicators will manage communications between quadtrees. Yes, because it is true the above analogy with parallel architectures, but it also true that, to resolve globally the Poisson problem on the totality of the considered domain, each grid will have to exchange, sooner or later, some data with the grids concerned with this sharing.

Looking at Figure 4.9b, for example, F and B must share data vital for the boundary conditions (for F) and to complete the numerical scheme for the cells of B close to those covered by F, that will not be considered in the assembly process of the final system to be resolved (see Figure 4.10).
Background octants needed by the numerical scheme (point P) and covereded by foreground grids will be replaced in the final system by an interpolation of the foreground octants around it.
Vice versa, the boundary values needed by foreground octants, being outside their own borders, will be obtained by an interpolation of the background octants reaching them (point P). More on interpolations will be told in the next chapter, exposing related examples.



(a) 5 grids (b) 2 grids

Figure 4.9: Example of not perfectly superimposed grids. As it can be seen, in fact, F3 and F do not exact match the border of the background grid B, less fine. F1, F2 and F4 are three other foreground grids, whose edges correspond to those of B

The system of unknown which is necessary to resolve to obtain a numerical

solution, is solved using PETSc and its Python binding PETSc4Py. PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. it is designed with an object-oriented style, and almost all user-visible types are abstract interfaces with implementations that may be chosen at runtime.
Those objects are managed through handles to opaque data structures which are created, accessed and destroyed by calling appropriate library routines. PETSc consists of a variety of components, each of which manipulates a particular family of objects and the operations one would like to perform on these objects. These components provide the functionality required for many parallel solutions of PDEs.
It employs the MPI standard for all message-passing communication.

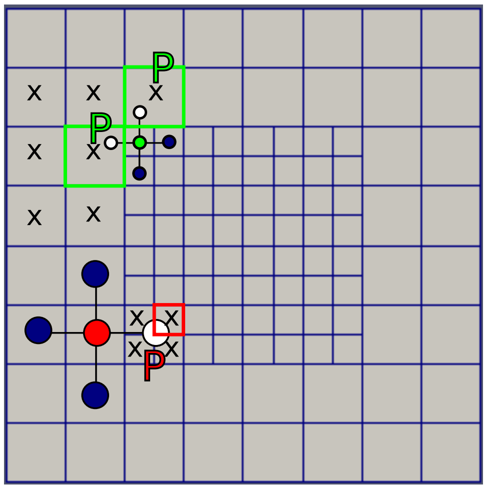


Figure 4.10: An example of interpolations needed at the edge between foreground and background: P is the center (the white point) required to complete the stencil of the red point on the loose grid, and it will be obtained interpolating the refined cells close to him. The two P, by contrast, are the centers (again, represented by two white points) requested to finalise the foreground stencil, and obtained interpolating the neighbouring cells of the background

Considering Figure 4.9a, the monolithic system will have the following form

$$\begin{bmatrix} D & R & R & R & R \\ P & D & 0 & 0 & 0 \\ P & 0 & D & 0 & 0 \\ P & 0 & 0 & D & 0 \\ P & 0 & 0 & 0 & D \end{bmatrix} \times \begin{bmatrix} x \\ x \\ x \\ x \\ x \end{bmatrix} = \begin{bmatrix} f \\ f \\ f \\ f \\ f \end{bmatrix}$$

in which

- Blocks D are the discretization matrices onto each grid (**parallel *MPI* intra-communications**). These block are assembled by each grid, without the participation of data originating from other grids (colored intra-communicators in Figure 4.11).

- Blocks R and P are the restriction and prolongation operators between the grids (**parallel *MPI* inter-communications**). Restriction blocks are assembled using data from foreground grids, whil prolongation operators are built with data from background grid (black intra-communicators in Figure 4.11).

- *MPI Datatypes* are used to exchange informations between the grids, which do not know each other, to fill the monolithic system. As previosuly said, in fact, It is desirable to communicate also structures or objects, without breaking them into individual arrays. But Why? Because, first of all, breaking data encapsulation complicates the code. And secondly, multiple communication operations result in higher overall communication cost, than one operation used with the same amount of data (because of the start-up latency).
  MPI provides two mechanisms that can be used on heterogeneous machines, and one of these it is exactly creating MPI derived datatypes (the other one is packing/unpacking data). Practically speaking, is equivalent to create a struct filled with user data, and then send it between processes.

The monolithic system is finally solved by PETSc, using the *MPI.COMM_WORLD* (see Figure 4.11, red enumeration) communicator.

4.7 Summary

How to resolve numerically a physical problem? What is a PDE? How to overcome it, knowing that it does not have an analytical solution? And

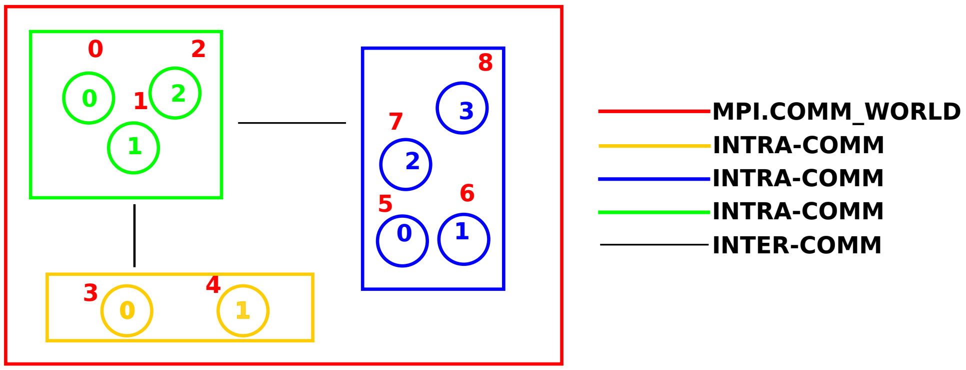


Figure 4.11: Inter and intra communication scheme, with global (red) and local (other colours) enumeration for the MPI processes at play. Intercommunicators for each group are depicted following the corresponding colors of local processes' enumeration. In this case, local group green can communicate with the other two, but them can not exchange messages, having not created the inter-communicator between them

what to do if the resolution domain is not a classical Cartesian grid?
To all these questions this chapter has tried to answer, but not only: the Poisson equation has been introduced, along with Python mechanisms to apply serialization on user defined data, required to communicate between multiple processes and to solve, in parallel, massive HPC problems, using the MPI paradigm (despite it is not the only one available in Python offering remote concurrency).
Furthermore, a practical example of the programming work done at Python level has been showed, using the Cythone extensions written *ad hoc*.
Last but not least, the algorithmic and programming approaches to the overlapping generic quadrilateral octree patches method, used to solve the 2D Poisson equation, has been outlined.

Bibliography

[1] Fenics project. `https://fenicsproject.org/pub/tutorial/sphinx1/`, . web page.

[2] Mpi4py. `https://mpi4py.readthedocs.io/en/stable/overview.html`, . web page.

[3] Python docs. `https://docs.python.org/3/`, . web page.

[4] Mpi4py. `https://mpi4py.readthedocs.io/en/stable/`, . web page.

[5] File formats for vtk, . Extract Taken from The VTK User's Guide.

[6] Love Hakansson Mikael Mortensen Rahman Sudiyo Berend van Wachem Bengt Andersson, Ronnie Andersson. *Computational Fluid Dynamics for Engineers.* Cambridge University Press, 2011.

[7] T. J. Craft. Basix finite volume methods, 2010. School of Mechanical Aerospace and Civil Engineering, University of Manchester.

[8] M. Peric J. Ferziger. *Computational Methods for Fluid Dynamics.* Springer, 2002.

[9] Argonne National Laboratory. *PETSc Users Manual.* 2017.

[10] Alfio Quarteroni. *Numerical Models for Differential Problems.* Springer, 2009.

[11] M Schafer. *Computational Engineering - Introduction to Numerical.* Springer, 2006.

[12] Nathan L. Gibson Vrushali A. Bokil. Finite difference, finite element and finite volume methods for the numerical solution of pdes, 2007. DOE Multiscale Summer School.

Chapter 5

Numerical Results

DDFV Discrete Duality Finite Volume

In this chapter we will show some numerical results obtained with our implementation.

The paragraph concerning this topic is however postponed to the sections relative to:

- the math explanation of some interpolations used in the examples to come;
- the explanation of the theory occurring in literature, in respect of the theoretical results governing the order of the interpolations on the frontier areas of the different quadtree patches;
- the theoretical results present in literature that guarantee the stability and the uniqueness of the solution for the Poisson equation.

The chapter will be followed by the conclusions covering all the work done, and that will pick up the threads of the topics covered, and of the issues not yet completed.

5.1 Border and Inner Interpolations

The interpolations presented in this section are classified according to the following criteria:

- interpolations used inside the domain of each quadtree. Here, only the diamond stencil is set out, because it is the finite-volume "counterpart" of the already covered (see Subsection 4.2.1) and well known centered finite-difference method;

- inteprolations used on the boundary area between background and foreground meshes. Here, two approaches are "rolled out": the classic bilinear interpolation, and the least squares method with the *pseudo-inverse* matrix usage.

5.1.1 Finite Volume Diamond Stencil

Since the early 2000's a new family of Finite Volume numerical methods has been developed, under the common name of *Discrete Duality Finite Volume* (**DDFV**) schemes.
In [2] and [5] they were introduced to study the Laplace equation on a large class of 2D distorted meshes, adding some unknowns to the problem; in particular, requiring unknowns on both vertices and centers of *primal control volumes.* In this way it was possible to obtain a full approximation of the gradient.
DDFV is a method oriented to this kind of reconstruction, and it uses the term *dual control volumes*, as opposed to the primal ones, and the term *diamond mesh.*
In this subsection, a simplification of such methods is presented, leaving aside the construction of the dual meshes, focusing instead on the diamond cells. A complete description of the DDFV scheme can be found in [9] and [7].
The need to use this kind of method is due to the use of octree meshes, which have within them local refining, requiring appropriate stencil to evaluate the fluxes around the edges.
And considering Figure 5.1, we can easily derive why the interpolation is called in this way: in fact, the quadrangle considered $ACBD$ is undoubtedly, a non degenerate diamond.
A point of information about the "new" unknown introduced by the DDFV mehods: in our simplification, considering always Figure 5.1, points C and D, which could be new vertices' unknowns, they will not be inserted in the overall system to be resolved, since they will be inteprolated using the closer centers' unknowns, thereby adding no extra "cost" to the resolution of the discretized equation.

If $A = (a_1, a_2, a_3)$ and $B = (b_1, b_2, b_3)$, we have that $\overrightarrow{AB} = (b_1 - a_1, b_2 - a_2, b_3 - a_3)$ (and, vice versa, $\overrightarrow{BA} = (a_1 - b_1, a_2 - b_2, a_3 - b_3)$).
By adding furthermore the following definition of **discrete gradient**

$$\overline{\nabla}(\phi) \quad \text{t.c.} \quad (x^{'} - x) \cdot \overline{\nabla}(\phi) = \phi(x^{'}) - \phi(x)$$

we have that

$$\phi_B - \phi_A = \frac{\partial \phi}{\partial \epsilon} AB_\epsilon + \frac{\partial \phi}{\partial \eta} AB_\eta$$

$$\phi_C - \phi_D = \frac{\partial \phi}{\partial \epsilon} DC_\epsilon + \frac{\partial \phi}{\partial \eta} DC_\eta$$

from which

$$\begin{aligned}
\frac{\partial \phi}{\partial \epsilon} =& \frac{\phi_B DC_\eta}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon} - \frac{\phi_A DC_\eta}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon} - \\
& \frac{\phi_C AB_\eta}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon} + \frac{\phi_D AB_\eta}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon} \\
\frac{\partial \phi}{\partial \eta} =& \frac{\phi_B DC_\epsilon}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon} - \frac{\phi_A DC_\epsilon}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon} + \\
& \frac{\phi_C AB_\epsilon}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon} - \frac{\phi_D AB_\epsilon}{AB_\epsilon DC_\eta - AB_\eta DC_\epsilon}
\end{aligned}$$

On a uniform octree, the previous evaluation of the derivatives "degenerates" automatically in the well suited and known form:

$$\frac{\partial \phi}{\partial \epsilon} = \frac{\phi B - \phi_A}{AB_\epsilon}$$

$$\frac{\partial \phi}{\partial \eta} = \frac{\phi C - \phi_D}{DC_\eta}$$

5.1.2 Bilinear Interpolation

In mathematics, bilinear interpolation is an extension of linear interpolation for interpolating functions of two variables (e.g., x and y) on a rectilinear

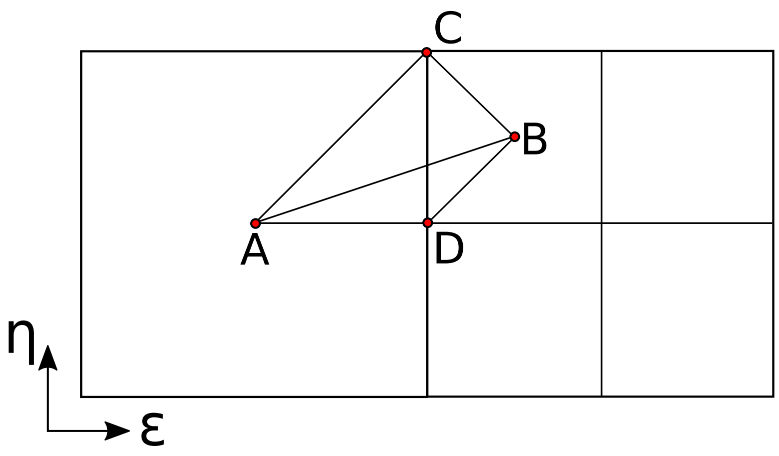


Figure 5.1: Diamond stencil on a 2:1 octree balance

2D grid.
The key idea is to perform linear interpolation first in one direction, and then again in the other direction. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location.

Considering Figure 5.2, $\phi(x,y)$ can be expressed in terms of $\phi_{i,j}$, leading to the following formulation:

$$\phi(x,y) = b_{1,1}\phi_{1,1} + b_{1,2}\phi_{1,2} + b_{2,1}\phi_{2,1} + b_{2,2}\phi_{2,2} \tag{5.1}$$

where coefficients $b_{i,j}$ are obtained comparingt the earlier equation with this one:

$$\phi(x,y) = \frac{\begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix}}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} \phi_{1,1} & \phi_{1,2} \\ \phi_{2,1} & \phi_{2,2} \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}$$

which can also be written as:

$$\begin{bmatrix} b_{1,1} \\ b_{1,2} \\ b_{2,1} \\ b_{2,2} \end{bmatrix} = \underbrace{\left(\begin{bmatrix} 1 & x_1 & y_1 & x_1 y_1 \\ 1 & x_1 & y_2 & x_1 y_2 \\ 1 & x_2 & y_1 & x_2 y_1 \\ 1 & x_2 & y_2 & x_2 y_2 \end{bmatrix}^T \right)^{-1}}_{A} \begin{bmatrix} 1 \\ x \\ y \\ xy \end{bmatrix}$$

being in fact

$$A = \frac{1}{(x_1 - x_2)(y_1 - y_2)} \begin{bmatrix} x_2y_2 & -y_2 & -x_2 & 1 \\ -x_2y_1 & y_1 & x_2 & -1 \\ -x_1y_2 & y_2 & x_1 & -1 \\ x_1y_1 & -y_1 & -x_1 & 1 \end{bmatrix}$$

From the last system we obtain the following explicit formulation for the

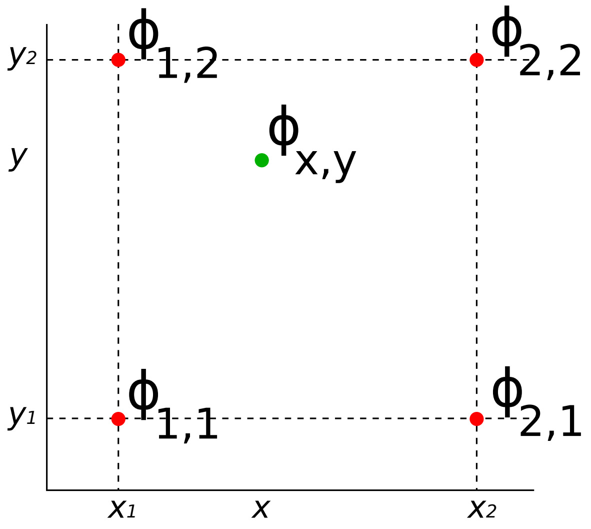


Figure 5.2: The four red dots show the data points and the green dot is the point at which we want to interpolate the data points, using the bilinear interpolation.

searched coefficients:

$$b_{1,1} = \frac{x_2y_2 - y_2x - x_2y + xy}{(x_1 - x_2)(y_1 - y_2)}$$

$$b_{1,2} = \frac{-x_2y_1 + y_1x + x_2y - xy}{(x_1 - x_2)(y_1 - y_2)}$$

$$b_{2,1} = \frac{-x_1y_2 + y_2x + x_1y - xy}{(x_1 - x_2)(y_1 - y_2)}$$

$$b_{2,2} = \frac{x_1y_1 - y_1x - x_1y + xy}{(x_1 - x_2)(y_1 - y_2)}$$

Referring to Equation 5.1, its derivatives are

$$\frac{\partial \phi\left(x,y\right)}{\partial x} = \sum_{i=1}^{2}\sum_{j=1}^{2}\frac{\partial b_{i,j}}{\partial x}\phi_{i,j}$$

$$\frac{\partial \phi\left(x,y\right)}{\partial y} = \sum_{i=1}^{2}\sum_{j=1}^{2}\frac{\partial b_{i,j}}{\partial y}\phi_{i,j}$$

5.1.3 Least squares

The method of least squares is a standard approach in regression analysis[1] to approximate the solution of overdetermined systems, i.e., sets of equations in which there are more equations than unknowns. "Least squares" means that the overall solution minimizes the sum of the squares of the residuals made in the results of every single equation.

Considering (x_i, y_j) as the points of the input data, we want to find a function ϕ such that it approximates the sequence of the given points.
This can be achieved minimizing the euclidean distance between the two sequences y_i and $\phi\left(x_i\right)$, namely

$$S = \sum_{i=1}^{n}\left(x_i - \phi\left(x_i\right)\right)^2 \tag{5.2}$$

(as already told, from here it derives the name "least squares").

In practical cases, $\phi\left(x\right)$ is parametric, and therefore the problem boils down to determine the parameters which minimize the distance of the points from the curve[2].
So, given the parametric formulation of the curve ϕ

$$\phi\left(x\right) = p_1\phi_1\left(x\right) + p_2\phi_2\left(x\right) + \cdots + p_k\phi_k\left(x\right)$$

[1]In statistical modeling, regression analysis is a set of statistical processes for estimating the relationships among variables.

[2]Of course it is necessary an amount of points equal or greater to the number of parameters from which depends the curve.

we define $\|r\| = \|Ap - y\|$, which is equal to $\|r\| = \sum_{i=1}^{n} \phi(x_i) - y_i$, being

$$A = \begin{bmatrix} \phi_1(x_1) & \cdots & \phi_k(x_1) \\ \vdots & \ddots & \vdots \\ \phi_1(x_n) & \cdots & \phi_k(x_n) \end{bmatrix}, \quad p = \begin{bmatrix} p_1 \\ \vdots \\ p_k \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

and we consider its square $\|r\|^2$, that is exactly what is written in Equation 5.2 (so, $\|r\|^2 = S$).
Therefore, the first step to minimize S is to examine its derivatives respect to the k parameters, and match them with 0:

$$\frac{d\|r\|^2}{dp_m} = 0, \quad \text{with} \quad 1 \leq m \leq k$$

The resolution of this system can be proven to be equal to resolve the system $(Ap - y)^T A = 0$, from which

$$p = \underbrace{\left(A^T A\right)^{-1} A^T}_{P_s} y \tag{5.3}$$

where P_s is the so called *Pseudo-inverse* matrix.

Assuming $\phi(x, y) = ax + by + cxy + d$, we will have that

$$A = \begin{bmatrix} x_0 & y_0 & x_0 y_0 & 1 \\ x_1 & y_1 & x_1 y_1 & 1 \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1} & y_{n-1} & x_{n-1} y_{n-1} & 1 \end{bmatrix}, \quad p = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}, \quad y = \begin{bmatrix} \phi_0 \\ \vdots \\ \phi_{n-1} \end{bmatrix}$$

which means, from Equation 5.3,

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = P_s \begin{bmatrix} \phi_0 \\ \vdots \\ \phi_{n-1} \end{bmatrix}$$

which gives us the parameters p in function of the values ϕ_n.

5.2 Accuracy and Order of Interpolation

An important question to ask when using composite grids is how to choose the order of interpolation so that the overall accuracy will be as good as the accuracy of the discretization formulae.
The answer to this question depends on the order of the PDE and the order of accuracy of the discretization formula.
Moreover, the answer also depends on the behaviour of the region of overlap as the mesh is refined.

Typically the overlap region will have a width which is approximately a constant times h, where h is a measure of the grid spacing. That is, the overlap region shrinks as the mesh is refined[3].

In Henshaw[4] it was shown that for solving second-order elliptic equations to second-order accuracy it is necessary to use third-order interpolation (quadratic interpolation) if the overlap between component grids decreases with h as the grids are refined. (Second-order interpolation, as linear interpolation, is sufficient if the overlap remains larger than some constant).

In Chesire and Henshaw[3], instead, it was shown how to choose the interpolation for a more general class of problems. Considering the model problem of solving a ($2p$)th-order boundary value problem on a one-dimensional composite grid, they have proved that when the overlap d decreases linearly with h, then the width of the interpolation formula, q, should be $2pr + 1$, where $2r$ is the order of acuracy of the discretization.
Thus, the width of the interpolation formula is the same as the width of the discretization formula.
If, on the other hand, d is a constant independent of h, then $q = pr + 1$.

To summarize the results of this section (see Figure 5.3), if

- $2p$: order of PDE (highest spatial derivative)
- $2r$: order of accuracy discretization
- q: width of interpolation formula

[3]Note: we will call an interpolant whose accuracy is $O(hP)$ to be a pth-order interpolant. In one dimension the standard interpolant on an equally spaced mesh which uses p points is a pth-order interpolant. Thus the standard linear interpolation (two points) is second-order interpolation while quadratic interpolation (three points) is third-order interpolation.

- d: width of overlap,

then

- $q = 2pr + 1$: width of interpolation formula if $d \propto h$
- $q = pr + 1$: width of interpolation formula if $d = O(1)$
- pr: number of interpolation points on each grid
- $d + pr(h_1 + h_2)$: total overlap.

This analysis can be related to the case when the composite grid equations degenerate to the standard central-difference approximation for a single grid. This situation occurs when $h_1 = h_2 = d$, in which case the interpolation points align exactly with grid points.
In this case, although the results previously cited said that interpolant should be ($2p + l$)-order accurate, classic finite-difference/finite-volume theory interpolation applies. What explained in this section can be easily ex-

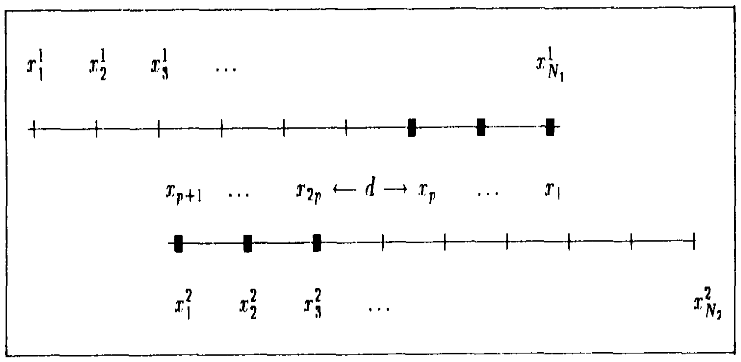


Figure 5.3: 1D composite mesh showing grid points x_i^k, discretization points x_i, and overlap d. $x_{N_1}^1$ represent discretization points of the first mesh G^1, while $x_{N_2}^2$ are the discretization points of the second mesh G^2. Instead, the points of interpolation are $x_j = x_{N_{1-j+1}}^1$, for $j = 1, \ldots, p$ and $x_{p+j} = x_j^2$, for $j = 1, \ldots, p$. Overla d is defined as the distance between the innermost interpolation point on G^1 to the innermost interpolation point on G^2: $d = x_p - x_{2p}$

tended to 2D cases, recalling that the previously one-dimensional theory indicates that, loosely speaking, the width of the interpolation formula should be equal to (or greater than) the width of the discretization formula in order to achieve an overall accuracy equal to the order of discretization.
This means that a 3 x 3 interpolation stencil (third-order or bi-quadratic interpolation) is required for second-order accuracy and a 5 x 5 interpolation stencil (fifth-order interpolation) is needed for a fourth- order accuracy.

5.3 Numerical Results

So, after having exposed some theoretical fundamentals, in this section we report a few numerical results.

5.3.1 First Outcome

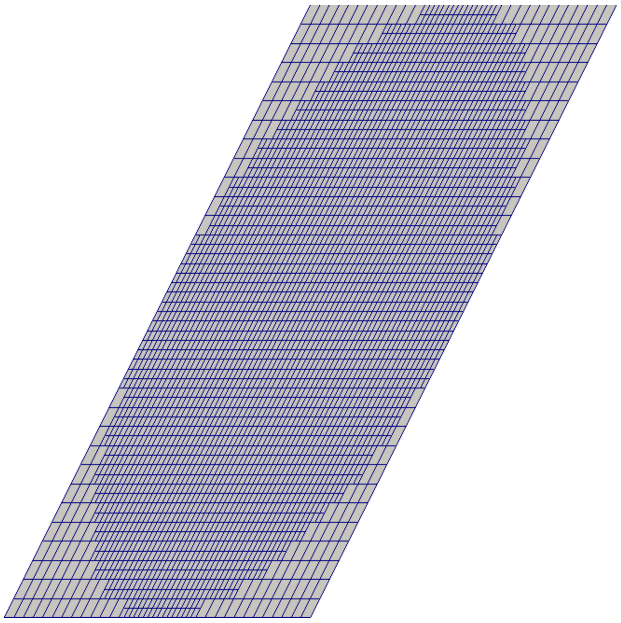
In this subsection we show the function $sin((x-0.5)^2+(y-0.5)^2)$ evaluated on two deformed grids: the first one (Figure 5.4) is a deformed single grid, with internal refinement automatically managed by the finite volume diamond stencil.
The second one, instead (Figure 5.5), is a deformed domain composed by two grids, whose distortions are, in no way, associated. Here, a classic finite difference central scheme is used, respectively on the background and on the foreground (one process each).

As we can see from the convergence orders of the errors reported in Figure 5.6, the finite-difference reconstruction (the 2nd one) obtains a first order decrease, due to the use of different sizes of the octants in the classic central scheme.
This can be outlined considering the following Taylor series expansion approximations to the second derivative, for non uniform grids:

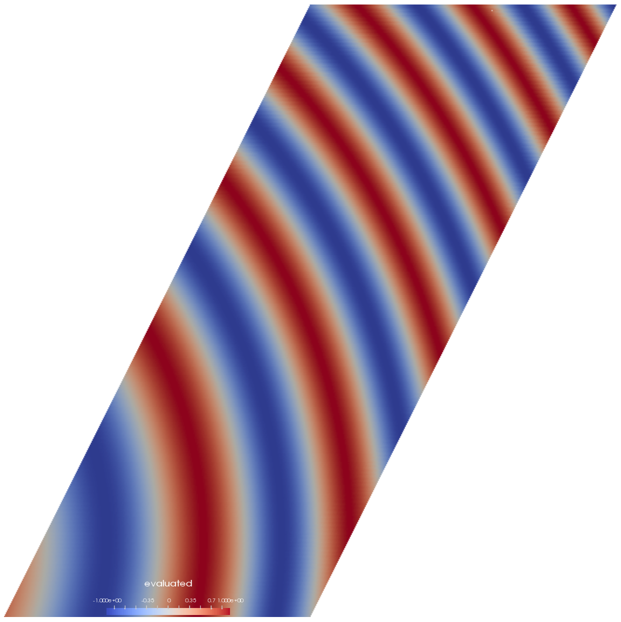
$$\left(\frac{\partial^2\phi}{\partial x^2}\right)_i = \frac{\phi_{i+1}(x_i-x_{i-1})+\phi_{i-1}(x_{i+1}-x_i)-\phi_i(x_{i+1}-x_{i-1})}{\frac{1}{2}(x_{i+1}-x_{i-1})(x_{i+1}-x_i)(x_i-x_{i-1})} -$$

$$\frac{(x_{i+1}-x_i)-(x_i-x_{i-1})}{3}\left(\frac{\partial^3\phi}{\partial x^3}\right)_i + H$$

where the leading truncation error term is first order, but vanishes when the spacing between the points is uniform, making the approximation second-order accurate.

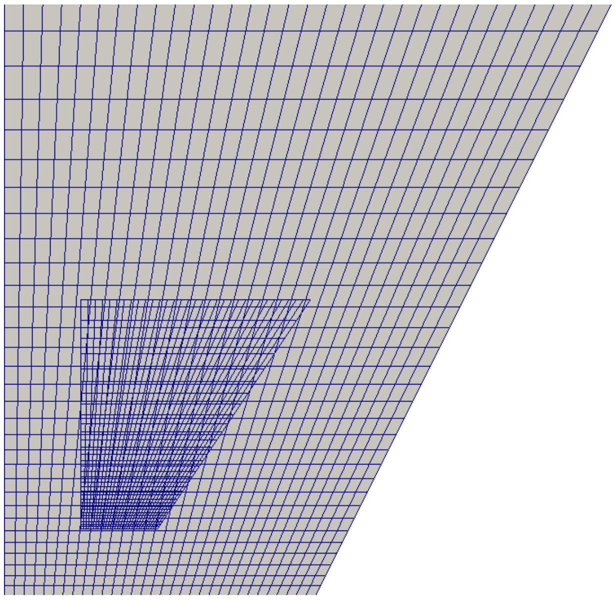


(a) Computational mesh

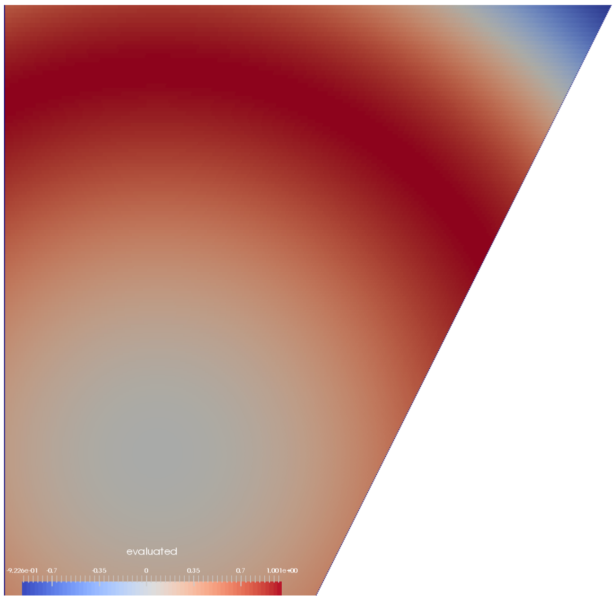


(b) Numerical reconstuction

Figure 5.4: Finite volume approximation, using a diamond stencil



(a) Background and foreground meshes



(b) Numerical reconstuction

Figure 5.5: Finite difference approximation, where for the evaluation of the unknown necessary for the background and covered by the foreground, a second order bilinear interpolation has been used; vice versa, to interpolate unknowns used by the foreground stencils, but outside its borders (and so, falling into the backgrounds), a second order approximation with a quadratic polynomial is used, interpolated with the least squares method.

However, because of the different sizes of "h", characteristic of both the grids, along the edges of the foreground no matter we are using second order interpolations, we will obtain a first order convergence, in accordance with Subsection 5.2, precisely because of the different characteristic quantities involved.
If, by the way, the composite grid degenerate into a "single" uniform one, then we have confirmed the results exposed in the already cited Subsection and in the previous Taylor expansion.

Finite volume approach (the 1st one) instead, using a simplified diamond scheme for the reconstruction of the fluxes on the interfaces, takes into account the presence of non uniform cells in the neighbours, and it is wery well suited for this kind of problems.

Its chart $\frac{\text{error}}{\text{refinements}}$ is reported in Figure 5.6b.

The result is actually reported to show the correctness of this scheme, although applied on a deformed and refined domain, composed by just one grid.

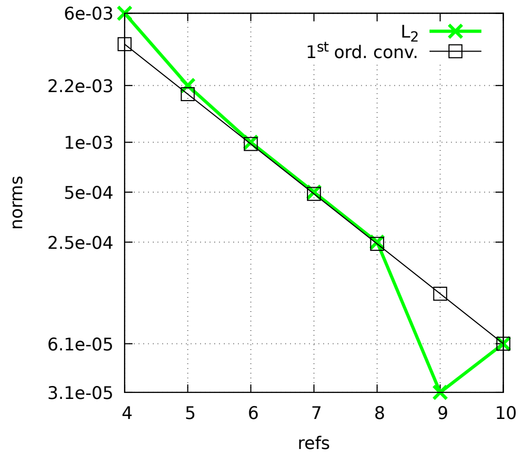
5.3.2 Multiprocesses

In this subsection, always the function $sin((x-0.5)^2+(y-0.5)^2)$ is reconstructed, but on two different domains, having in common the fact to be splitted on four processes (two for each grid). Figures 5.7a and 5.9a indicate visually the different components of the comprehensive numerical domain, coloring each one by a different color.
In the last case of this subsection is presented the case concerning the cinfiguration presented in Figure 4.9a. Only the results for the F2 grid are reported, having the other grdis the same trend.

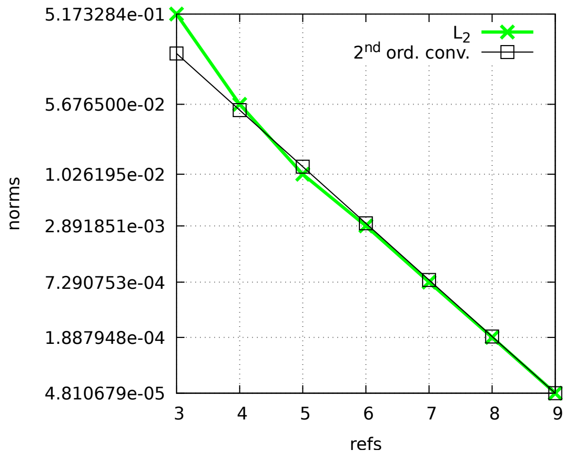
5.3.3 Heat Equation

The heat equation is a parabolic partial differential equation that describes the distribution of heat (or variation in temperature) in a given region over time.
For a function $u\,(x,y,t)$ of two spatial variables and the time variable t, the heat equation is

$$\frac{\partial u}{\partial t}=\alpha\Delta u$$

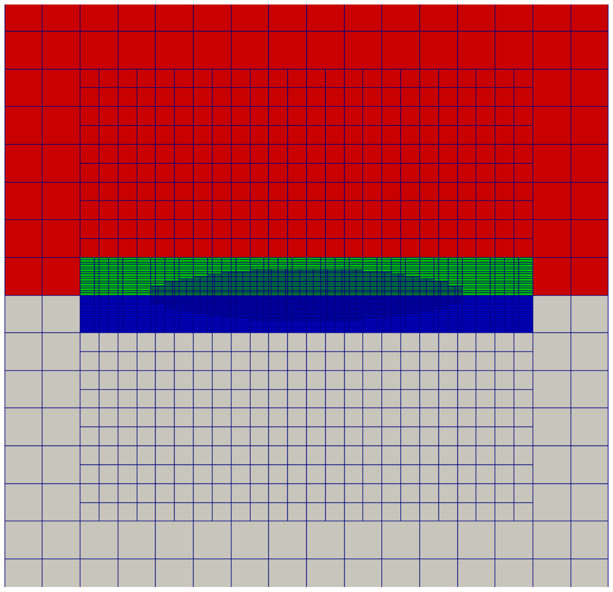


(a) Error $||L_2||$ degrowth for the finite difference case of Figure 5.4

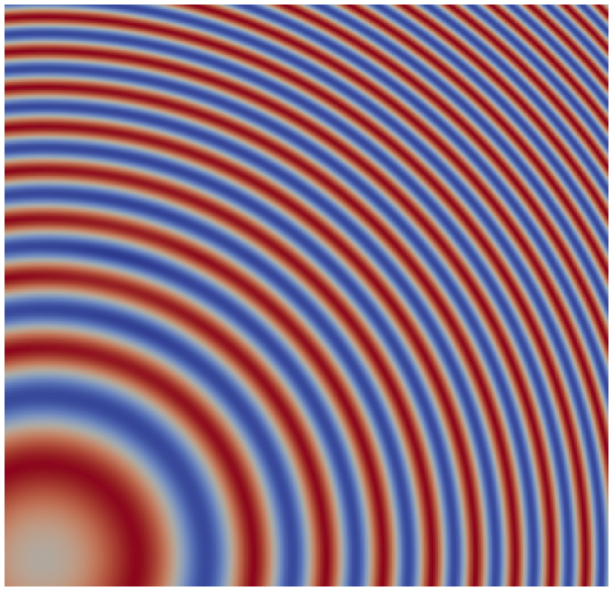


(b) Error $||L_2||$ degrowth for the finite volume case of Figure 5.5

Figure 5.6: On x axis are reported the refinement levels (for the finite difference case, just of the foreground grid, but for the background the trend is analogous), while on the y axis is shown the difference between the exact solution and the numerical approximation, in norm L_2

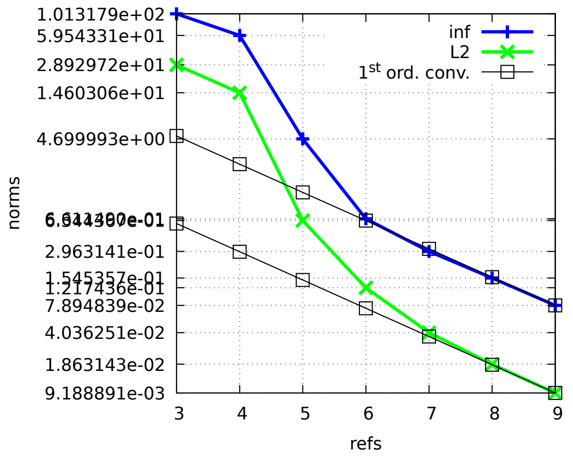


(a) Domain decomposition between four processes: green and blue are for the foreround, red and grey for the background

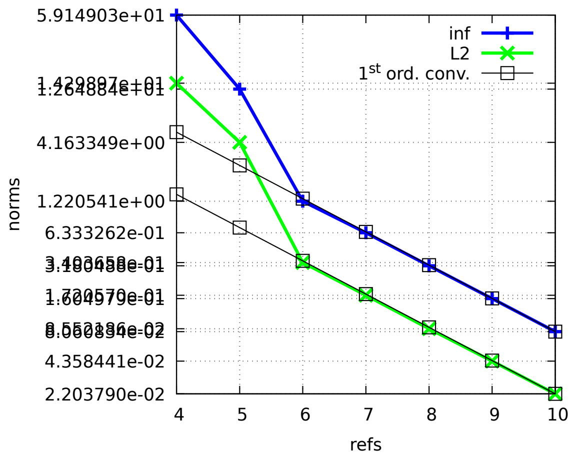


(b) Numerical approximation for $sin((x-0.5)^2+(y-0.5)^2)$ on the domain above

Figure 5.7: Each grid is refined internally, and the octants of the background covered by the foreground will not be considered in the monolithic system build to resolve the problem in its entirety. The foreground represents a buffer zone around an hypothetical object

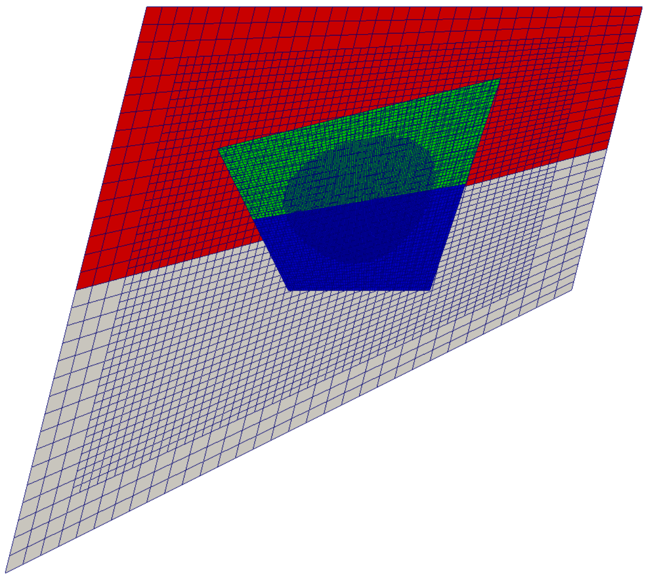


(a) Background error's convergences

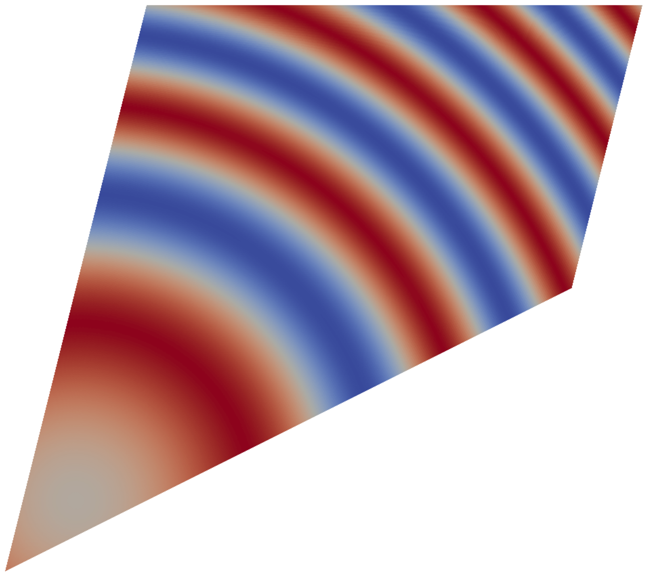


(b) Foreground error's convergences

Figure 5.8: Behaviour of the error made on th numerical approximation. Norms L_2 and ∞ are reported, both for the background and the foreground. As expected, first order degrowth is obtained

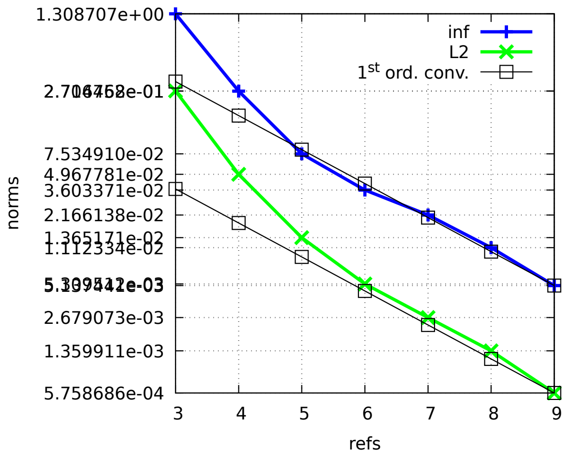


(a) Domain decomposition between four processes: green and blue are for the foreround, red and grey for the background

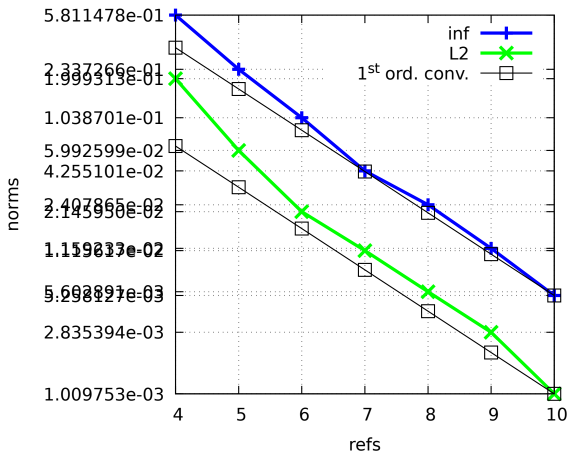


(b) Numerical approximation for $sin((x-0.5)^2+(y-0.5)^2)$ on the domain above

Figure 5.9: Each grid is refined internally, and the octants of the background covered by the foreground will not be considered in the monolithic system build to resolve the problem in its entirety, as already said in Figure 5.7

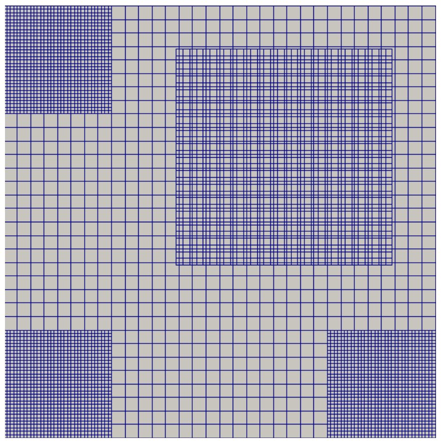


(a) Background error's convergences



(b) Foreground error's convergences

Figure 5.10: Behaviour of the error made on th numerical approximation. Norms L_2 and ∞ are reported, both for the background and the foreground. As expected, first order degrowth is obtained


(a) Domain decomposition between five processes, one for grid

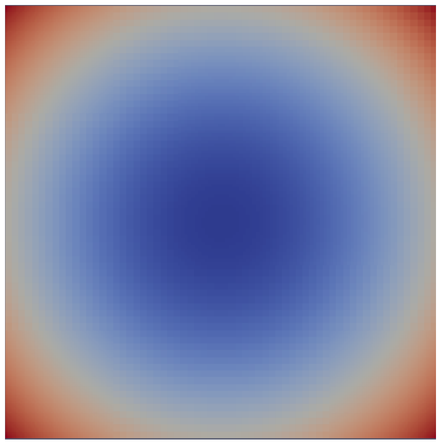
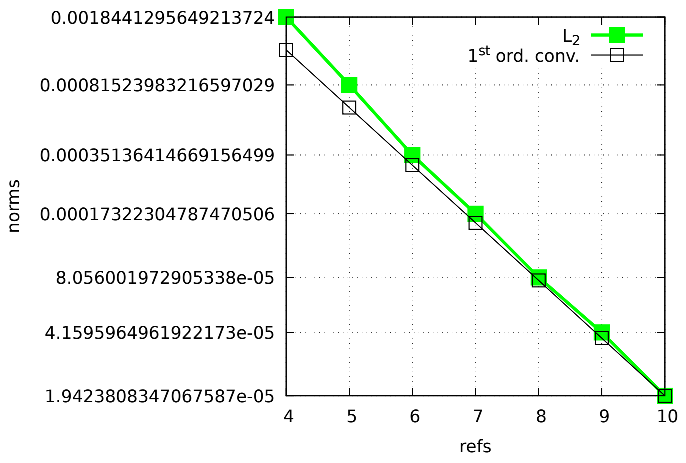
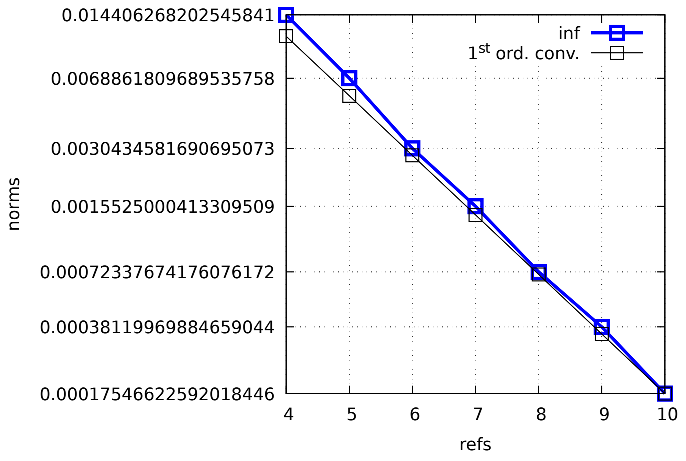

(b) Numerical approximation for $sin((x-0.5)^2+(y-0.5)^2)$ on the domain above

Figure 5.11: Each foreground grid represents a refinement for the background, also showing not perfectly superimposed refinings



(a) Foreground error's convergence, L2 norm



(b) Foreground error's convergence, infinity norm

Figure 5.12: Behaviour of the error made on th numerical approximation. Norms L_2 and ∞ are reported, just for the F2 grid in Figure 4.9a, having the other grdis the same trend.

where α is the thermal diffusivity[4]. For the mathematical treatment it is sufficient to consider the case $\alpha = 1$, and of course supplement an initial condition $u\left(x, y, 0\right) = f\left(x, y\right)$ and the boundary conditions.

For the example reported in Figure 5.13, we have chosen a *backward time, centered space* (*BTCS* scheme, that lead to the following discretization, for the time step m and for spatial position i, j, of the previous equation:

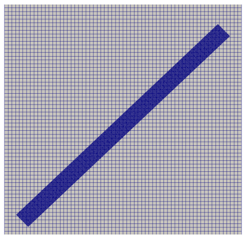
$$\frac{u_{i,j}^{m} - u_{i,j}^{m-1}}{\Delta t} = \alpha \frac{u_{i-1,j}^{m} + u_{i+1,j}^{m} - 4u_{i,j}^{m} + u_{i,j+1}^{m} + u_{i,j-1}^{m}}{\Delta x^2} + O\left(\Delta t\right) + O\left(\Delta x^2\right)$$

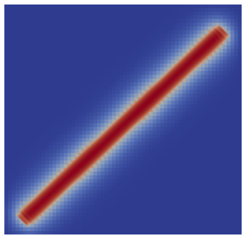
Implementation of the BTCS scheme requires solving a system of equations at each time step but, being unconditionally stable (as we have seen in the previous chapter), it presents a huge advantage over the *FTCS* (*forward time, centered space*) approach, to obtain stable solutions.

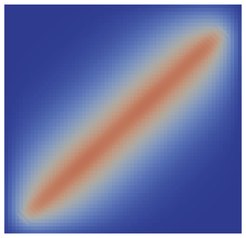
5.4 Summary

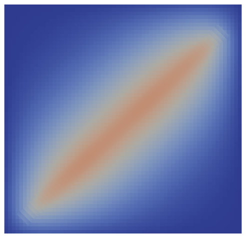
If the previous chapter has dealt in detail the numerical and programming approaches and techniques used in the course of my work, this chapter has emphasized the visualization of the results, showing furthermore the convergence orders obtained.
Before that however, some pages were spent to line up the readers on the interpolation schemes used, on the theoretical results against which we bumped, and on the background of theory necessary to proceed in conducting the tests.
The overall conclusions, by the way, are postponed in the next chapter.

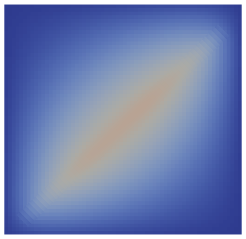
[4]In heat transfer analysis, thermal diffusivity is the thermal conductivity k, divided by density ρ and specific heat capacity at constant pressure c_p: $\frac{k}{\rho c_p}$. It is the measure of thermal inertia, which means that in a substance with high thermal diffusivity, heat moves rapidly through it because the substance conducts heat quickly relative to its volumetric heat capacity or “thermal bulk”.

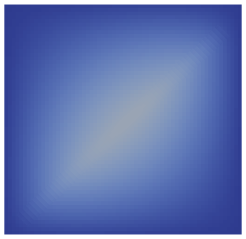

(a) Computational Domain

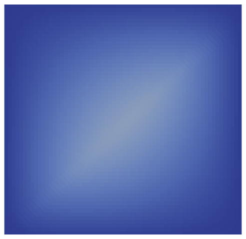

(b) Time step = 1


(c) Time step = 10


(d) Time step = 20


(e) Time step = 40


(f) Time step = 80


(g) Time step = 100

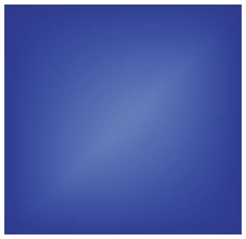

(h) Time step = 160

Figure 5.13: Heat distribution on a computational domain depicted by Subfigure 5.13a. The foreground grid, a stretched rectangle, has been imposed with an initial condition of $T = 1$, while the background has had the homogeneous $T = 0$ condition, both for the initial and boundary constraints. As we can see, as time passes, global temperature tends to comply, reaching a uniform value

Bibliography

[1] N. Nikiforakis W.D. Henshaw A.R. Koblitz, S. Lovett. Direct numerical simulation of particulate flows with an overset grid method. *Journal of Computational Physics*, 2017.

[2] K. Domelevo and P. Omnes. A finite volume method for the laplace equation on almost arbitrary two-dimensional grids. *M2AN*, 2005.

[3] W. D. Henshaw G. Cheshire. Composite overlapping meshes for the solution of partial differential equations. *Journal of Computational Physics*, 1989.

[4] W. D. Henshaw. Thesis, 1985. Department of Applied Mathematics, California Institute of Technology, 1985 (unpublished).

[5] F. Hermeline. A finite volume method for the approximation of diffusion operators on distorted meshes. *J. Comput. Phys.*, 2000.

[6] A. M. Wissink S. R. Kohn Hornung, R. D. Managing complex data and geometry in parallel. *Engineering with Computers*, 2006.

[7] Stella Krell. Stabilized ddfv schemes for stokes problem with variable viscosity on general 2d meshes. *hal-00385687v1*, 2009.

[8] Pablo Ouro Fermin Navarrina Sofiane Khelladi Ignasi Colominas Luis Ramirez, Xesus Nogueira. A higher-order chimera method for finite volume schemes. 2017.

[9] Giulia Lissoni Thierry Goudon, Stella Krell. Numerical analysis of the ddfv method for the stokes problem with mixed neumann/dirichlet boundary conditions. *hal-01502397*, 2017.

[10] D. A. Hysom R. D. Hornung Wissink, A. M. Enhancing scalability of parallel structured amr calculations, 2003. Proceedings of the International Conference on Supercomputing 2003 (ICS‘03), San Francisco.

Chapter 6

Conclusions

Conclusions are never easy because, as their name suggests, they represent the end of something.
And finishing something, it is never entirely a good thing. Every experience, every work, every "path", they contain good and bad memories, and their mingle usually leave us with a feeling of sour looking back, of sad, even if until the day before the "longed-for" conclusion we were thrilled by the change.

And this is exactly the sense that I feel rereading my thesis, because at first glance it brings to my memory all the people I have met during this period, and knowing that I won't see them on a daily basis let me feel bad. Secondly, it reminds me all the efforts made to do what it has been shown, and all the jitters when things did not came back.

And with these memoirs, thoughts surface to my mind, concerning all the tasks pre-fixed and that eventually, for a lack of time or for exhaustion, they were not accomplished.

In the list of these "unfinished business", first of all, I would state the absence of the 3D implementation, which would have surely given an "impact" more pronounced to this work, while sticking around the 2D cases could make it feel like a "toy" developing.
Another thing, for which I get upset with myself, is that an hybrid approach has not been coded, leaving empty that space always saved for an MPI+OpenMP or MPI+MPI combination (by the way, calling *pragmas* is possible in Cython, just disabling the G.I.L., and MPI shared memory windows are binded into *mpi4py*).

And of course, mathematically speaking, a case more complicated than the Poisson equation could have been considered; maybe the *Stokes flow*[1] which, being a steady and linearized form of the Navier-Stokes equations, is a very well studied and even popular equation in the CFD universe.

But should have, could have, do not matter. A Ph.D. thesis is something that does not end; simply, it is interrupted at some point, as someone wiser than me said once.
And besides, usually there is an explanation for all; also because, otherwise, what has been said so far would not explain the already cited feelings of melancholy that I perceive, having been exposed just “not accomplished” (yet) aspects of this thesis.

Then, why do not mention the work done to get a working Python API for PABLO. In Chapter 4 we have seen just some extracts from the code written, but the efforts are deeper.
Just think about the conversion of C++ templates in Cython templates, which is something you usually do not find in books. Or what about the usage of derived classes from a C++ context into a Python one. And this again, is something you have to figure it out, to avoid errors, in Python, concerning double free or corruption, being obliged to delete the pointers both of the derived and the parent classes.
And pointers, writing a Python wrapper, are something you should be concern about. For example, a typical pointer’s conversion is like

```
darray3 getCenter(Octant* octant)
...
...
def get_center(self               ,
               uintptr_t idx      ,
               ...)
    ...
    ...
    center = self.der_thisptr._getCenter(<Octant*><void*>idx)
    ...
```

where `get_center` is the wrapper for the original function `getCenter`.
Once you know how to cast the pointers, parameter passing is something realtivily easy but again, you have to figure it out. In this case for example,

[1]Stokes flow, also named *creeping flow*, is a type of fluid flow where advective inertial forces are small compared with viscous forces. Being the Reynolds number low, this is a typical situation in flows where the fluid velocities are very slow and the viscosities are very large.

there is a double change: the `uintptr_t`[2] integer to `void` pointer conversion first, and later the `void` to the `desired_type` pointer cast.

Continuing citing the essays spent in the generation from scratch of the code, having already cited (Chapter 4) the changes applied to the `build_ext` class of the module `Cython.Distutils`, a word should be spent on the usage of MPI inter and intra communicators, which ususally leave room to the common and unifying *MPI_COMM_WORLD*, thereby avoiding the additional complexity due to the construction of an inter-communicator from two intra-communicators.
Indeed, this operation will require separate collective operations in the local group and in the remote group, as well as a point-to-point communication between a process in the local group and a process in the remote group.
In this regard, a specific method was written to create **automatically** the MPI intercommumicators for the different grids, depending on the number of the grids inserted.
And this method has to deal with the choice of local and remote "peer" communicators, and has to choose a safe tag to ensure the uniqueness of the connection. The function `MPI_INTERCOMM_CREATE` (Python corresponding function is `Create_intercomm`), in fact, can be used to create an inter-communicator from two existing intra-communicators, in the following situation: at least one selected member from each group (the group leader) has the ability to communicate with the selected member from the other group; that is, a "peer" communicator exists to which both leaders belong, and each leader knows the rank of the other leader in this peer communicator.
Furthermore, members of each group know the rank of their leader.

And of course, speaking of MPI, we should consider the non trivial Python to MPI data conversion, necessary to avoid reduntant overheads in passing data between processes.
For example, considering this code snippet

```
...
...
block_type = dtype('(1, 10)i4, (1, 1)f8, (1, 1)f8')
data = array(((1,2,3,4,5,6,7,8,9,10), 65.567, 3.0), \
              dtype = block_type)
block_length = [10, 1, 1]
block_displ = [0, 40, 48]
```

[2]It is an unsigned int that is capable of storing a pointer, which typically means that it's the same size as a pointer.

```
mpi_types = [MPI.INT, MPI.DOUBLE, MPI.DOUBLE]
mpi_d_t = create_struct(block_length,
                        block_displ,
                        mpi_types).Commit()
if (rank == 0):
data = numpy.zeros(1, block_type)
recv([data, mpi_d_t], source = 1, tag = 1)
print(data)
elif (rank == 1):
data = numpy.array(data, block_type)
send([data, mpi_d_t], dest = 0, tag = 1)
```

we have a practical example of what we said in Chapter 4, speaking about MPI Datatypes; here `mpi_d_t` will be used, together with `data`, to specify all the informations MPI needs:

- the number and types of all the data members/fields;
- the relative offset of the fields from the beginning of the structure;
- The total memory occupied by a structure.

As for the programming part, I would like now to underline some choices and problematics that have affected the mathematical and dimensional contexts.

For the 3D case, its absence is explained by having tried to obtain a second order method, focusing our strengts and attentions on implemeting different ways and methodologies of interpolations at the borders, passing from a classic finite-difference centered scheme to several finite-volume ones, characterised by the commonality of the diamond stencil (as seen in Chapter 5), essential starting point to use the intrinsic refining capacity of the Octrees. Although these essays have given us a discreet knowledge of the problem and of all its issues, we still have finished to meet those constraints previously exposed by the theory sections of the preceding chapter, not being able to ensure, everywhere at the interfaces, a 3 x 3 interpolation stencil, considering just one halo ring of neighbours around the affected cells of the grids.
This "constraint" on the thickness of the ring was imposed because, at the time the experiments were being carried out, ther was not in PABLO a function to enlarge that, and modify a so complicated library was out of the scope of this thesis.
Besides, evaluation of additional "outher halo points"[3] could have lead to a

[3]Each procesor has a domain consisting of a core region surrounded by a ring of "inner halo points". The outher halo points are points which lie in another processors calculation

communications increase.

So, here we are, to take stock of the efforts by summarising what was, and not, done.

I am honestly and deeply proud of what has been accomplished, because I think it is something not yet proposed, something “new” in the approach made towards the resolution of the problems considered, and something that effectively and in practical terms shows a concrete application using different Python’s modules, which previously could have look just as an end in itself as of a uselfuness purely theoretical, in a community (that of Python’s users) which has always had no business with modelling and simulations.
But not only new usages of Python’s module were shown: a completely new Python module has been developed, PABLitO, which is, for what I know, the first Python API to use a parallel balanced linear octree in an interpreted way.

And speaking more personally, for me, this professional interlude in my life did really matter, letting me meet students, researchers, professors, very well versed in their subjects (and this is crucial for the occupational growth of one individual), and allowing me to do what I definitely want to (and what I am actually doing): working in the applied mathematics field, but under a profile more IT oriented. And my supervisors let me “carte blanche”, putting always, in front of everything, my intent, my aspirations, my desire to investigate certain and specific topics.

And their will to always value firstly the interests of their doctoral students, myself included, has established in me a great sense of recognition for them. For this I thank them, and I want to carry out that famous checklist of “unfinished business” not only for the Python community; not only for me.
I want to continue this project, on my own time, yes for both myself and to contribute to the growth of free software, but for them too. Because, right now, I can assure them that their teaching will not be lost, but I also want to show them that their trust has not been wasted.
Because they deserve this as chefs, as persons, as mentors.

domain, and the inner halo points are those required by other processors.