https://lib.uliege.be https://matheo.uliege.be

Optimizing NESTOR, a tool for preparing meteorological and surface data for the MAR model

Auteur : Grailet, Jean-François
Promoteur(s) : Fettweis, Xavier
Faculté : Faculté des Sciences
Diplôme : Master en sciences géographiques, orientation global change, à finalité approfondie
Année académique : 2021-2022
URI/URL : http://hdl.handle.net/2268.2/14780

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

Optimizing NESTOR, a tool for preparing meteorological and surface data for the MAR model

By JEAN-FRANÇOIS GRAILET

Faculty of Sciences

Department of Geography

UNIVERSITY OF LIÈGE - BELGIUM

A master thesis submitted for the
MASTER'S DEGREE IN GEOGRAPHICAL SCIENCES, GLOBAL CHANGE ORIENTATION

Members of the jury:
XAVIER FETTWEIS (supervisor)
SÉBASTIEN DOUTRELOUP
NICOLAS GHILAIN

JUNE 2022

Abstract

Simulating the Earth's climate is a complex and challenging task that the scientific community is actively researching with various approaches. Besides the computer models that simulate the climate for the entire Earth's surface, also called **G**lobal **C**limate **M**odels (GCMs), a significant part of the ongoing research focuses itself on **R**egional **C**limate **M**odels (RCMs). These computer models simulate the climate over high resolution grids modelling specific regions of the globe in order to produce finer results w.r.t. GCMs, notably regarding precipitations (i.e., rainfall and snowfall). The `MAR` model is a good example of RCM, and is particularily effective at simulating precipitations as well as snow and ice. As such, several research groups actively use it to study the evolution of polar regions or to predict future hydroclimatic conditions in specific regions of the world.

In order to be as realistic as possible, a RCM needs to feed the borders of its grid with pre-existing meteorological data in order to take account of how the climate evolves outside of the region of interest. This data, or forcing fields, can come either from real-life measurements or from predictions computed by a GCM. In the case of the `MAR` model, the large scale grids must first be processed by `NESTOR`, a companion software that is responsible for downscaling said grids, i.e., inferring high resolution data from the input domain to initialize the regional grids and prepare the forcing fields.

While the `MAR` model has been maintained and updated on a regular basis since its creation, `NESTOR` has not received a major update since 2004, though some components have been added much more recently to meet the needs of `MAR` users. While it is still doing its intended task, `NESTOR` requires a significant amount of time to process most of its typical use cases, a problem which constitutes an additional constraint to `MAR` users.

This master thesis thoroughly reviews the source code of `NESTOR` to identify its main issues and subsequently introduces a few simple changes that significantly improve its performance. When all suggested changes are applied, typical use cases can be computed in (much) less than one minute, with some of these use cases having been accelerated up to 40 times with respects to the unedited `NESTOR`. A comparison of the results of a `MAR` simulation started with the old `NESTOR` with those of a second simulation kickstarted with the output files produced by the updated `NESTOR` demonstrates that the suggested changes did not alter the results of the `MAR` model, despite the well-known chaotic nature of some atmospheric processes. Finally, the updated `NESTOR` also drops obsolete features with respects to the previous version and slightly improves the readability of its source code.

Keywords: forcing fields, surface fields, interpolation, regional model, `NESTOR`, `MAR`

ACKNOWLEDGEMENTS

First of all, I would like to thank Prof. Xavier Fettweis for paving my way towards climate science despite my unconventional initial background, i.e. computer science, but also for giving me the opportunity to work so soon on a tool that is regularly used by researchers at the Laboratory of Climatology and Topoclimatology of the University of Liège. As my long-term career goal is to work on the more technical aspects of computer climate models, being able to dive into the subject even before finishing my one-year master's degree has been both instructive and rewarding.

I would also like to thank, more generally, all the professors and teaching assistants who taught me during this very special academic year and despite my numerous (and occasionally naive) questions. I am especially grateful to Prof. Louis François, who taught me throughout the first semester in basically all the courses I was attending then. As a Physics professor, M. François has indeed been very patient with me despite the fact that I have had very few Physics courses during my Computer Science cursus at the Faculty of Applied Sciences.

Last but not least, I would like to thank my family, friends and former colleagues (including my former Ph. D. advisor Prof. Benoit Donnet) who proved to be very supportive of my unusual choice, as a young doctor in computer science, to complete one additional master's degree in climate science. As the ongoing climate change is set to be the issue with the most significant impact on our daily lives for the coming decades, becoming able to contribute to climate research as a computer scientist was very important to me. Planning such a change was not without its challenges, but I have most probably tackled those more easily thanks to the support of my relatives and my peers.

Author's declaration

This master thesis was written using the *University of Bristol Thesis Template* [2] created by V. Breña-Medina, and under the license *Creative Commons CC BY 4.0* [6], with very slight modifications.

TABLE OF CONTENTS

LIST OF TABLES

List of Figures

Chapter 1

Introduction

Simulating the Earth's climate is a complex and ambitious task featuring multiple challenges. To begin with, computer climate models must accurately account for all the physical processes that occur in the atmosphere and at the interface between the atmosphere and the surface (ground or sea) [13, 14, 20, 23]. Another major challenge lies in accurately modelling the Earth's surface and atmosphere. The surface can typically be modelled by a two-dimensional grid, with each dimension accounting for the longitude and the latitude, respectively. In such a grid, a cell will correspond to a small area of the Earth's surface whose corners will have given longitude and latitude coordinates. To account for the Earth's atmosphere, a third dimension can be added to model the air column above each cell in the initial two-dimensional grid (which will be subsequently referred to as ***surface cell***) [13, 23]. The final three-dimensional grid is depicted in Figure 1.1.

Naturally, a challenge with this modelling approach lies in choosing the right resolution for the grid, i.e., determining how much surface area will be covered at most by each surface cell, as there is a trade-off between the model resolution and the overall computational cost of the simulation. The number of vertical levels in each air column is also a parameter, though the order of magnitude of the Earth's surface area far exceeds that of the thickness of the first layers of the atmosphere. Indeed, the troposhere, where most of the weather processes occur, is only a dozen kilometers thick on average [11, 14, 20], while the Earth's surface area is more than 500 millions of square kilometers [36].

On the one hand, if a surface cell models a large area of the Earth's surface (e.g., 1° of longitude and 1° of latitude per pixel, leading to a 360 by 180 grid [1]), the computer model will be able to simulate the climate over extended periods of time. On the other hand, cells modelling smaller areas of the Earth's surface may not only allow the model to produce finer results, but may also help it simulate more complex physical processes: for instance, if a single cell models an area of one square kilometer,

[1]It is worth noting such a resolution means each cell does not cover exactly the same area in square kilometers.

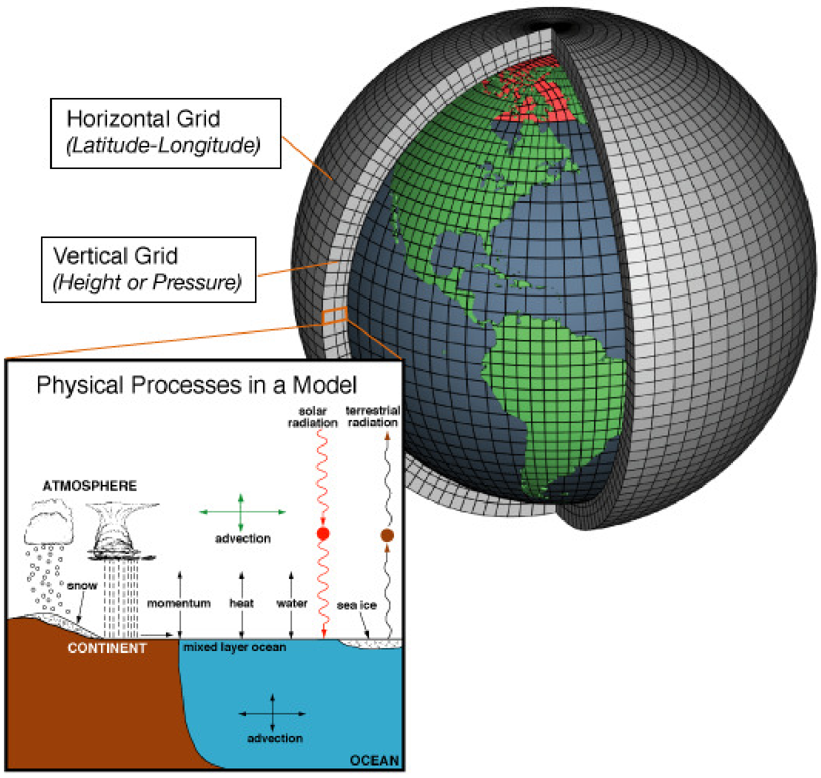


Figure 1.1: Schematic view of the typical 3D modelling of the Earth's surface and atmosphere [24].

the computer model may be able to estimate surface wind turbulence. Of course, using a higher resolution also comes at the cost of increasing the overal computational cost of the simulation, which also means a high resolution model may be only used to simulate short periods of time.

To tackle this resolution challenge, various computer models have attempted to simulate the Earth's climate at smaller scales, i.e., without simulating its whole surface. In some models, the Earth's atmosphere is even modelled as a single air column, such as MacKay and Khalil's radiative convective model, which they used in the early 1990s to simulate how the Earth's atmosphere would react to varying concentrations of greenhouse gases and varying cloud parameters [19]. More recently, the research community has focused on the development and use of **R**egional **C**limate **M**odels (or RCMs [2]) [8, 18]. These models aim at simulating the Earth's climate over a specific region of its surface with a high resolution grid to produce finer results, notably regarding precipitations (i.e., rainfall and snowfall). The `MAR` model (for ***M**odèle **A**tmosphérique **R**égional*) is such a model, and has been developped and used since the early 2000's by various research groups [21]. The `MAR` model is especially well known for its exhaustive and refined simulation of rainfall, snow and ice (both snowfall and snow/ice layers), and as such, has been used to study the evolution of polar regions [1, 12, 16] as well as hydroclimatic conditions of other regions of the world, notably to study flood risks [43].

Because the climate of a specific region can still be influenced by atmospheric processes occurring

[2]RCM is also an acronym for **R**adiative **C**onvective **M**odel. In this work, RCM always denotes a regional model.

at a large scale, such as the North Atlantic oscillation [11, 40], a RCM still has to take account of how the climate evolves outside of the region of interest to produce credible results. A common approach to achieve this consists of feeding the cells at the boundaries of the regional grid with real meteorological data or prospective data produced by a Global Climate Model (or GCM), i.e., a climate model simulating the Earth's entire surface and atmosphere. This process is known as ***boundary forcing***. Because of the difference in resolution between the meteorological/GCM data and the regional grid, the former must first be ***downscaled*** in order to be used by the latter. In climate modelling, ***downscaling*** refers to the process of inferring high resolution data from a grid with a low resolution, and is typically achieved with the help of interpolation algorithms [27, 33–35, 38].

Both grid initialization and boundary forcing of the `MAR` model are managed with its own companion software, `NESTOR` [22]. `NESTOR` stands for ***NEST****ing* ***OR****ganization for the preparation of meteorological and surface fields* and, as its full name implies, is designed to process existing data (in the form of NetCDF files [39]) to prepare the initial grid of the `MAR` model as well as the values that will be used for boundary forcing throughout the simulation. A schematic view of how `NESTOR` processes a large scale grid into a regional grid that is suitable to the `MAR` model is provided by Figure 1.2.

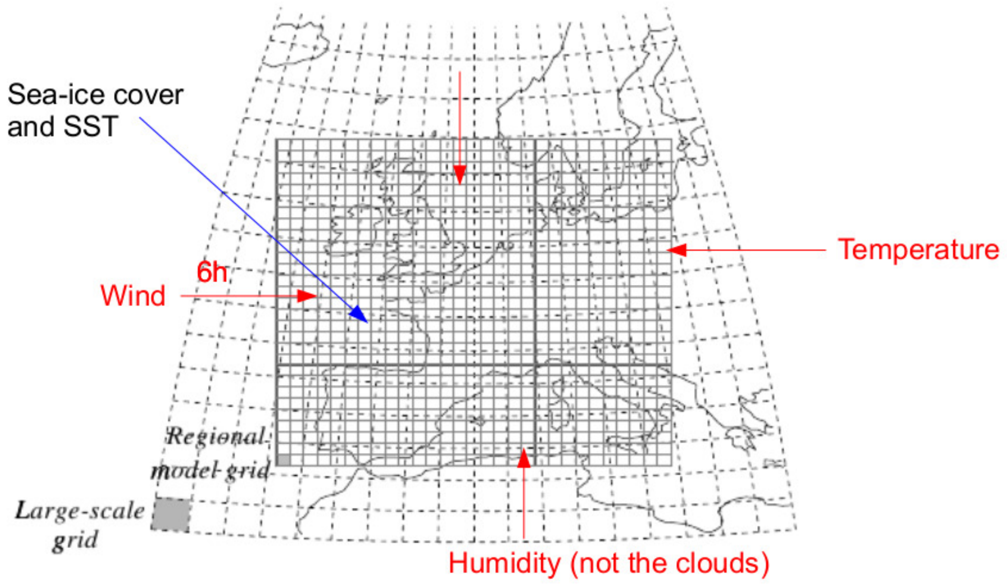
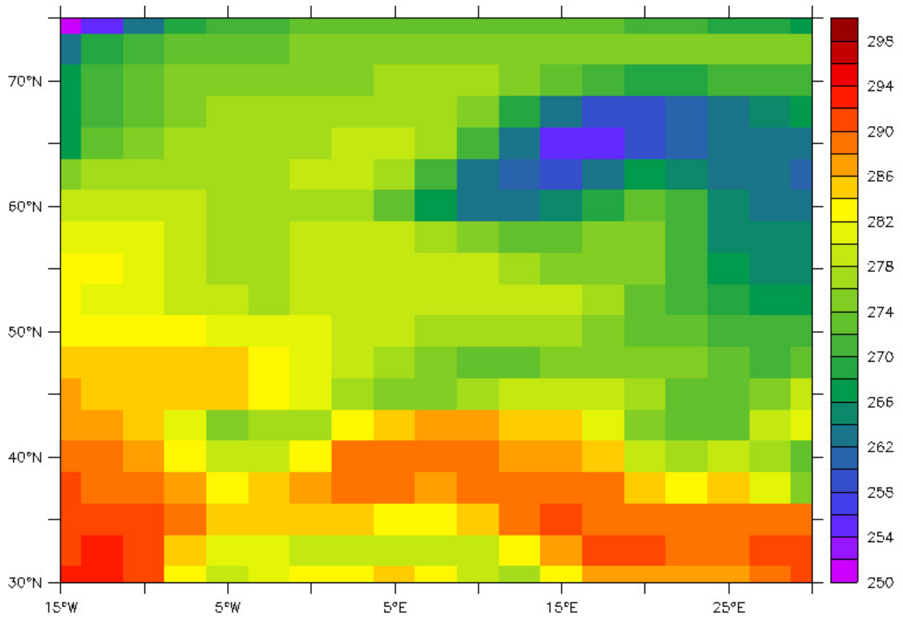


Figure 1.2: Schematic view of how `NESTOR` processes a large scale grid into a regional grid [10].

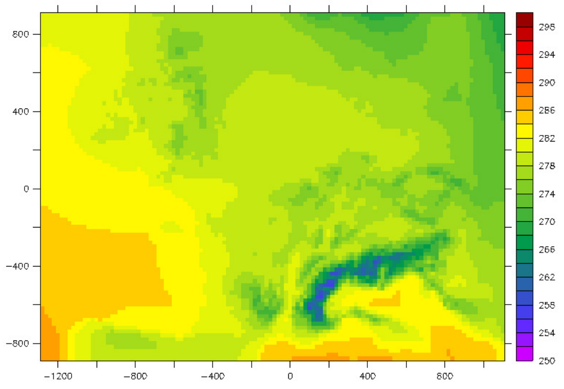
To produce regional grids, `NESTOR` uses bilinear [34] and bicubic [33] interpolation for the longitude/latitude axes, and makes use of linear [38] and cubic [35] interpolation for the air columns above each surface cell. Parts of the implementation of such methods re-use Fortran code from the 1992 reference book "*Numerical Recipes in FORTRAN 1977*" by Press et al. [27]. Figure 1.3 provides practical examples of a grid from an input domain being interpolated into regional grids by `NESTOR` [3], with Fig. 1.3a depicting a 2D surface temperature map from the large scale input domain, Fig. 1.3b providing a regional 2D surface temperature map over Western Europe fit for the `MAR` model obtained

[3]Using the original version, i.e., prior to any modification presented in this work.

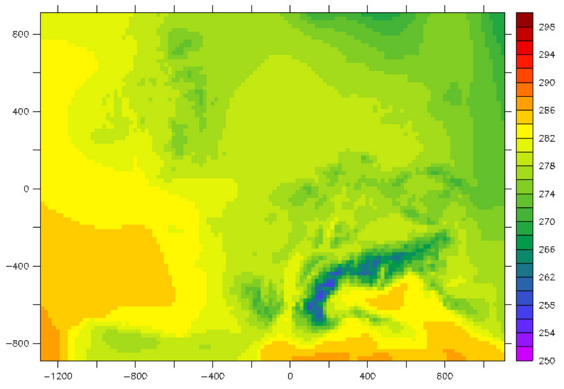
through bilinear interpolation, and Fig. 1.3c providing a similar map obtained through bicubic interpolation. The initial map (Fig. 1.3a) provides surface temperatures at noon on the first of December 2021 in the NCEP-NCAR Reanalysis 1 dataset [25]. This dataset normally provides temperatures for the entirety of Earth's surface, but for a convenient comparison, Fig. 1.3a is zoomed on Europe. The same temperature scale, in Kelvin, is used in every map for readability. The resolution of the NCEP-NCAR grid is 2.5 by 2.5 degree of longitude/latitude, while the resolution in both Fig. 1.3b and Fig. 1.3c is 20 by 20 kilometers per cell. Note how Fig. 1.3c provides smoother area curves with respect to Fig. 1.3b, which results from using bicubic rather than bilinear interpolation.



(a) Surface temperature (zoomed on Europe) on December 1st, 2021 (NCEP-NCAR R1 [25])



(b) Bilinear interpolation over Western Europe



(c) Bicubic interpolation over Western Europe

Figure 1.3: Examples of input and output of NESTOR. All figures provide surface temperature maps for December 1st, 2021 at noon, all using the same temperature scale (250 to 300 Kelvin) for readability.

Because of its relationship with the MAR model, NESTOR has been designed and written in Fortran around the same time as the very first version of the MAR model. As such, some comments found in specific sub-routines of NESTOR mention said sub-routines were first written as soon as 1996 [22]. However, NESTOR has not been maintained as thoroughly as the MAR model itself: while the latter has been continuously updated and improved over time until very recently, the last major changes brought to the core code of the former seem to date back to 2004. Indeed, several major interpolation sub-routines, as well as some additional components, are commented with the NESTING keyword and a date comprised between 2000 and 2004. This does not mean that NESTOR has not been updated since then: a few components of its current code are much more recent, such as the HWSDsl and GLOcov sub-routines (meant to prepare a regional grid for the soil texture and soil cover, respectively) which are dated to 2018. Some additions in specific sub-routines are also dated to various years of the 2010-2019 decade. Nevertheless, the source code of the early 2022 NESTOR appears as old, especially given that it is still formatted in fixed-formed Fortran [4].

Of course, having old-looking code does not prevent a computer program from doing its intended job, and NESTOR is still regularly used due to its output being a prerequisite to run the MAR model. However, the main issue currently experienced by MAR users with NESTOR is its slow execution time: despite having straightforward tasks to carry out (downscaling existing maps into regional maps for the MAR model), NESTOR can take from a few minutes to around half an hour to prepare a single month of MAR input data with bilinear interpolation, knowing that using bicubic interpolation is even slower. This means that a researcher planning a multi-year MAR simulation has to wait for several hours for the complete MAR input to be ready, or has to start the MAR simulation when the first input files are ready while NESTOR is still running in background to prepare the subsequent MAR input. In either case, NESTOR is cumbersome: in the former scenario, NESTOR delays the start of the MAR simulation by several hours, while it adds an additional burden to the host machine in the latter case.

Interestingly, and as will be commented later in this manuscript, a few parts of NESTOR code have been partially optimized: running NESTOR with ERA-5 data [9] is significantly faster than running it with NCEP-NCAR data [25]. Nevertheless, NESTOR, as of January 2022, can be considered as a serious candidate for code optimization. The purpose of this master thesis is therefore to thoroughly review the source code of NESTOR to find the sub-routines or sets of instructions where it needlessly loses time and to refactor them, all while ensuring that the output of the refreshed NESTOR does not change the MAR output despite the well-known chaotic nature of meteorological processes [26], i.e., the risk of observing different outcomes following minor decimal changes.

To tackle this task, this master thesis will be articulated around four main questions.

1. **Where does early 2022's NESTOR waste time ?** Before optimizing NESTOR, its typical execution scenarios should be profiled, and its main flaws carefully characterized. Such an analysis will serve as a basis for optimizing NESTOR. This first question will be addressed in Chapter 2. This chapter will also mention a few obsolete features which were highlighted and eventually removed from NESTOR throughout this master thesis.

[4]which notably enforces a limit of 72 characters per instruction, a remnant of the early days of the language [37].

2. **How can `NESTOR` be sped up where it loses time ?** A solution to tackle each of the flaws identified in Chapter 2 will be presented. Potential improvements over some of these solutions will also be discussed. This second main question will be tackled in Chapter 3.

3. **How fast is `NESTOR` after optimizing its code ?** The execution time of the improved `NESTOR` will be compared with the one delivered by the initial version, that is, as of January 2022, in order to quantify how much faster the former is. This third question will constitute the first part of Chapter 4.

4. **Is the `MAR` sensitive to the output produced by the improved `NESTOR`? What about its sensitivity to the improved bicubic interpolation ?** Because of small decimal changes between both `NESTOR` versions, which will also be shortly discussed in Chapter 4, the impact of the new `NESTOR` on the `MAR` model must be thoroughly assessed. More generally, the sensitivity of the `MAR` model will be assessed, as the impact of choosing the (refactored) bicubic interpolation over the classical bilinear interpolation for a same region and time period will also be discussed. The impacts of the new `NESTOR` on the `MAR` model will constitute the second part of Chapter 4.

After discussing each aforementioned question, a general conclusion will be provided in Chapter 5 to summarize the contributions of this master thesis and potential improvements that could be brought to `NESTOR` in the future.

Chapter 2

An in-depth review of the source code of NESTOR

This chapter reviews thoroughly the source code of NESTOR as of January 2022 [22], and as such, answers question 1 from Chapter 1. An overview of the source code is first provided (Sec. 2.1), followed by a short commentary on the modern use of NESTOR (Sec. 2.2). The main issues of NESTOR, i.e. parts of its code where it needlessly loses time, are subsequently characterized by analyzing and profiling practical use cases (Sec. 2.3). Finally, other issues, such as code readability or obsolete, even practically unusable features, will be shortly discussed (Sec. 2.4) before concluding this chapter with a roadmap of the modifications that should be brought to NESTOR (Sec. 2.5).

2.1 Overview of the source code of NESTOR

The unedited source code of NESTOR, i.e. as found within the `src/` directory of its typical archive as of January 2022 [22], consists in more than 90 files, among which 70 are Fortran source files (`.f`) and 19 are `.inc` files designed for sharing common variables and arrays across the entire program. Both numbers can be reduced to 69 and 17, respectively, by omitting library files (namely `libUN.f`, `libUN.inc` and `NetCDF.inc`, all used for reading NetCDF files [39] in Fortran). A few additional files feature unique extensions (such as `NESTOR.stereo` and `USRant.offset`) but appear to only provide variants of specific instructions or of given subroutines for the sake of providing a complete picture to future programmers, as they are ignored during compilation.

The vast majority of the Fortran source files of NESTOR (i.e., `.f` files without `libUN.f`) define and implement a single subroutine, which shares its name with the file itself (e.g., the `GLOcov.f` source file defines and implements the `GLOcov` subroutine). Occasionally, side subroutines or functions are also provided along the main subroutine, but are usually only called by the main subroutine of the file. As a result, the list of files that make up the source code of NESTOR already provides a clear view of its subroutines. It is worth noting that almost all subroutines of NESTOR exhibit a name restricted to six characters, which is again an old convention of fixed-form Fortran [37]. Within the list of the Fortran source files, three are especially important for understanding NESTOR.

- `NESTOR.f` provides the main body of instructions of NESTOR. This includes, in order: welcome and main error messages, handling the input parameters (as provided by the `NSTing.ctr` control file [1]), initializing all grids, and what will be referred to as the ***main loop***. This loop is responsible for generating the regional grids for the MAR model in the chronological order. Each of its iterations reads large scale grids stored in a NetCDF file [39] associated to the input domain at a given date and time and downscales them into regional grids that will be used by the MAR model to perform boundary forcing at a specific time step. It is worth reminding that the MAR model does not perform boundary forcing at every time step, but rather after a short period of time. By default, this period is set to 6 hours (and is also a parameter of NESTOR).
- `NSTint.f` provides the subroutine `NSTint`, which contains the sequence of instructions for downscaling the grids from the input domain associated to given date and time into regional grids. It can be roughly considered to constitute a single iteration of the main loop of NESTOR, though said loop calls other subroutines besides `NSTint`, notably to manage input/output operations (notably via `LSCinp` and `MARout`). Nevertheless, `NSTint` provides the core algorithmical operations of a single iteration of the main loop of NESTOR.
- `INTERp.f` provides the main interpolation subroutines involved in downscaling the input domain: `INThor` (horizontal interpolation for a 2D grid or one level of a 3D grid; calls another subroutine depending on the input parameters), `INTbil` (bilinear interpolation), `INTbic` (bicubic interpolation) and `INTlin` (linear interpolation; used for interpolating vertical levels).

It is worth noting that `INTbil` and `INTbic` from `INTERp.f` are tailored for ***regular grids***, i.e., grids where the difference in longitude/latitude between adjacent cells is constant, just like in most datasets covering the entire Earth's surface (e.g., as shown in Fig. 1.3a in Chapter 1). NESTOR also provides two subroutines for downscaling non-regular grids (e.g., produced by a past MAR simulation). `INTERp.f` provides the `INTnrg` interpolation subroutine for non-regular grids, but as will be discussed in Sec. 2.4.2, it is never called in practice by the early 2022's NESTOR, which rather uses the `INTnrg2` subroutine, provided by a separate file, whose downscaling algorithm is essentially a bilinear interpolation performed on a different grid format.

Outside of these three `.f` files, most of the subroutines of NESTOR are rather straightforward. In particular, a large number of subroutines are dedicated to managing a specific format of input data. For instance, all subroutines whose associated file name is suffixed in `vgd.f` (`vgd` is for $\underline{v}$ertical $\underline{g}$ri$\underline{d}$) build the air column above a given surface cell, the instructions of each such subroutine varying depending on how vertical levels are handled in the associated data source.

[1] as edited by the user or generated with a Shell script.

A considerable number of subroutines of `NESTOR` are also dedicated to its initialization steps, such as creating the output grid format (any subroutine whose file name ends with `hgd.f`, for horizontal grid) or preparing regional grids that are not subject to boundary forcing during a `MAR` simulation. For instance, the subroutines `GLOveg`, `CORveg` and `BELveg` are all three used to downscale known vegetation characteristics of the Earth's surface to the regional scale, and are only called at the beginning of a `NESTOR` run. Another example is `GLOcov`, which reads a high resolution NetCDF file providing soil cover for the entire Earth's surface (i.e., is a given surface cell covered by a forest, a city, a desert, etc.) to produce the regional equivalent for the `MAR` model. The same subroutine uses a second high resolution file to manage the ice cover upon preparing data over Greenland or Antartica.

2.2 Past and modern uses of `NESTOR`

Another noteworthy characteristic of `NESTOR` is its considerable number of options. Outside the basic parameters that are the start and end dates of the downscaled data, the data source and the preferred types of interpolation, early 2022's `NESTOR` also provides alternative modes to the usual downscaling (rain disaggregation and wind gust estimation) and numerous options for the initialization steps. As shortly mentioned in Sec. 2.1, it also provides the option of using `MAR` data as input, i.e., the results of past `MAR` simulation are used and downscaled in order to simulate the climate over a sub-region with an even higher resolution. For example, results of a simulation over the British Isles (United Kingdom and Ireland) can be downscaled to mainland Scotland, with surface cells going from a 12 by 12 to a 5 by 5 kilometers resolution.

Strangely enough, with the notable exception of being able to force the `MAR` model with pre-existing `MAR` data, several of these options are difficult if not impossible to use in practice: a quick look into `NESTOR.f` notably reveals that the wind gust estimation mode requires a `WGustE` subroutine which is absent from the current source code of `NESTOR`, and whose sole call has been commented. A complete inventory of these options will be provided later in Sec. 2.4.2.

Hopefully, most options are ignored by `MAR` users in practice. In fact, `NESTOR` is nowadays exclusively used for preparing `MAR` input data, despite technically allowing the preparation of regional grids for other purposes, as demonstrated by the existence of the subroutines `GRAhgd`, `GRAvgd`, `CPLhgd` and `CPLvgd` which are used to prepare regional grids for GrADS (**Gr**id **A**nalysis and **D**isplay **S**ystem [3]; `GRA` in `NESTOR`) and SVAT (**S**oil **V**egetation **A**tmosphere **T**ransfer) coupling (`CPL`), respectively.

As for the format of the input data, modern users of `NESTOR` typically feed it with data from:

- the ECMWF (**E**uropean **C**entre for **M**edium-Range **W**eather **F**orecasts), either with hybrid (`ECM`) or pressure (`ECP`) levels, such as ERA5 dataset [9],
- the NOAA (**N**ational **O**ceanic and **A**tmospheric **A**dministration), with NCEP-NCAR datasets [25],
- a past **G**lobal **C**limate **M**odel simulation (`GCM`),
- or a past `MAR` simulation (`MAR`).

In addition to these sources, NESTOR allows use of LMD data (***L**aboratoire de **M**étéorologie **D**ynamique*), though it can nowadays be considered as obsolete. In summary, modern NESTOR users exclusively use it to produce MAR input data, using data from the ECMWF (ECM and ECP), the NOAA (NCP), a GCM (GCM) or the MAR model itself (MAR). This leaves only one output format (MAR) and four different data sources. Finally, several options of NESTOR are typically left with default values by modern users: for instance, De Ridder and Schayes' soil model [7] is always used though users can actually turn it off. However, turning off this model results in two variables within the output data ending up completely empty [2]. This suggests that several features of NESTOR as used at the start of this master thesis should be either removed, either re-evaluated, as will be discussed in Sec. 2.4 and Sec. 2.5.

2.3 Inefficient operations in NESTOR (as of January 2022)

With the architecture of NESTOR and its typical modern use in mind, a handful of realistic use cases can be established to assess its performance (again, as of January 2022) but also to profile its execution and to find the subroutines where NESTOR wastes time the most. Sec. 2.3.1 first presents typical use cases and their performance. Sec. 2.3.2 subsequently discusses what makes these use cases slow or faster than others with the help of a profiling command line tool (`gprof` [4]). The main issues of early 2022's NESTOR are subsequently deduced and discussed: the setting of the vertical levels of the input domain (Sec. 2.3.3), soil texture and soil cover interpolation (Sec. 2.3.4), expensive or redundant interpolation subroutines (Sec. 2.3.5) and inefficient input domain management (Sec. 2.3.6).

2.3.1 Practical use cases of NESTOR and their performance

A simple way of building comparable use cases with NESTOR consists of running it for the same period of time over the input domain, e.g., each experiment will process a certain amount of days with the same dates. A realistic scenario consists of preparing MAR input data for 30 days of the same month and the same year. The same output domain, i.e., the same region on which the input domain is downscaled, can also be used in all scenarios. The experiments can then differ in two ways:

1. the data source (ECM/ECP, NCP or GCM) and the associated grid dimensions,
2. the type of horizontal interpolation (bilinear or bicubic).

Considering these options, a set of seven experiments has been designed, each experiment consisting of a single NESTOR run on 30 days.

Six first typical use cases have been first deduced by using the data sources ECM (e.g. ERA-5 [9]), ECP and NCP [25] with either bilinear or bicubic horizontal interpolation. Both ECM and ECP provide data for Europe and Northern Africa only, while NCP provides data for the entire Earth's surface. All six experiments involved data for December 2021. There is no GCM scenario because the subroutines used

[2]namely TS (soil temperature) and SW (soil wetness).

with such a data source are essentially the same as ECM, while there is actually different subroutines used upon using either ECM or ECP. A seventh use case, involving MAR on MAR forcing, can complete this set of experiments in order to be representative of the modern use of NESTOR. However, for this last experiment, the output regional domain could not be the same as for the six other experiments, due to the input domain (a simulation over the British Isles during July 2010 conducted in Fall 2021 [3]) already being relatively small. Nevertheless, this experiment vastly differs from the others anyway due to the unique interpolation method, which is already optimized in January 2022's NESTOR.

To get a sample execution time for each of these experiments, the Linux `time` command [5] has been used. For reminders, this command allows the user to evaluate how much time elapsed during the execution of any program. More precisely, at the end of the execution, it provides three values: `real`, `user` and `sys`. The `real` time corresponds to the wall clock time, while `user` provides the time spent on a CPU outside system calls, whereas `sys` gives the time elapsed on a CPU while performing system calls. Because all seventh experiments have been run on a server of `climato.be` (`srv5`) shared with other users (meaning the CPU could not always be used by NESTOR for its whole execution), and because `sys` times were always negligible (a few seconds), the `real` time will be selected as representative of the execution time of NESTOR in any of the seven use cases.

Input	**Domain**	**Grid dimensions**	**Interpolation type**	`user` **time**
ECM	Europe	367x167x40	Bilinear	1m 58' 85"
			Bicubic	54m 01' 66"
ECP	Europe	181x91x31	Bilinear	26m 58' 78"
			Bicubic	39m 28' 99"
NCP	Earth	144x73x18	Bilinear	12m 03' 74"
			Bicubic	19m 44' 01"
MAR	British Isles	100x100x24	Bilinear (variant)	2m 16' 79"

Table 2.1: Summary of 7 sample experiments carried with NESTOR on `srv5` from `climato.be`.

Table 2.1 provides the `user` execution times obtained with each of the seven sample NESTOR runs, along with the conditions of each. All experiments prepared input MAR data for a period of 30 days, the six first in the table using December 2021 data and downscaling on the same region (Western Europe, roughly centered on France, with surface cells of 20 by 20 kilometers). Dimensions are provided with the longitude axis first, followed by the latitude axis and the number of vertical levels. Finally, the MAR on MAR experiment downscaled the input domain on mainland Scotland with a 5 by 5 kilometers resolution (versus 12 by 12 kilometers for the input domain).

Table 2.1 demonstrates that, no matter the use case, NESTOR spends several minutes to fully prepare one month (30 days) of MAR input data. This holds especially true for the sample runs relying on bicubic interpolation to downscale 2D grids, and this also holds true for the use cases involving bilinear interpolation and the data sources ECP and NCP. Interestingly, the experiments with the data source ECM provide both extremes in terms of execution time for the whole of Table 2.1. Its

[3]in the context of the CLIM0017 course [10].

comparatively fast execution time while using bilinear interpolation can be explained by the fact that a subroutine used by NESTOR during this specific scenario has already been optimized, as will be detailed in Sec. 2.3.3. It should be noted that the variations in `user` times also appear to be related to the dimensions of the input domains: indeed, both experiments with NCP as data source completed faster than ECP with similar conditions but a smaller grid as for the input domain. The fact that the experiment with ECM and bicubic interpolation is almost an hour long can also be tied to the large dimensions of the input domain (largest grid among all data sources).

2.3.2 Profiling NESTOR in typical use cases with `gprof`

To better understand why NESTOR runs so slow in most experiments included in Table 2.1, the tool `gprof` [4] can be used. `gprof` is a programming tool that investigates the call graph of a program written in either C, Pascal or Fortran, i.e., how many calls to a specific subroutine or function have been made and approximately how much time has elapsed while they were used. Because NESTOR features a large number of subroutines and because `gprof` goes as far as enumerating underlying subroutines or functions called by the libraries, only the five most costly subroutines or functions, i.e. the ones in which NESTOR spent the most time, will be showed each time.

Listing 2.1: The five most called functions/subroutines in NESTOR with NCP/bilinear (by `gprof`)

```
  %    cumulative    self
time     seconds    seconds     name
48.15     354.39    354.39      netcdf_mp_nf90_get_var_3d_fourbytereal_
 5.09     391.83     37.44      nf_inq_var_
 4.47     424.76     32.93      nf_inq_dim_
 3.04     447.14     22.38      utf8proc_decompose
 2.84     468.04     20.90      utf8proc_decompose_char
```

Example 2.1 provides the five most called subroutines/functions used while running the NCP use case with bilinear interpolation, as provided by `gprof`. The given ranking, which is a rough simplification of the full output of `gprof`, provides four columns for each function/subroutine: a percentage of the total execution time spent with said function/subroutine, a cumulative time in seconds, the time spent with the function/subroutine itself in seconds, and its name. What immediatly catches the eye here is that all the listed functions are not NESTOR subroutines, but I/O functions that are either from the `NetCDF.inc` library (namely `nf_inq_var_` and `nf_inq_dim_`) either from underlying code, and whose names heavily imply they are used for reading NetCDF files and processing their content. In particular, the function prefixed with `netcdf_` took almost half of the entire execution time. This suggests that NESTOR wasted a considerable amount of time reading and processing NetCDF files, which calls for an optimization of its file reading strategy.

Example 2.2 provides a similar ranking to that of Example 2.1 for the experiment involving bilinear interpolation and the data source ECP. This second ranking obviously shows that the ECP/bilinear use case suffer from the same I/O issues as highlighted in Example 2.1: indeed, the functions are identical, only with slightly different values as for the first column (percentage of the total execution time) and greater times for the second and third columns. The latter observation can be tied to the larger dimensions of the `ECP` input domain w.r.t. the `NCP` input large scale domain (cf. Table 2.1).

Listing 2.2: The five most called functions/subroutines in `NESTOR` with `ECP`/bilinear (by `gprof`)

```
  %    cumulative    self
 time    seconds    seconds     name
49.40     801.49    801.49      netcdf_mp_nf90_get_var_3d_fourbytereal_
 5.15     884.98     83.49      nf_inq_var_
 4.57     959.15     74.17      nf_inq_dim_
 3.30    1012.68     53.53      utf8proc_decompose
 2.92    1060.01     47.33      utf8proc_decompose_char
```

Listing 2.3: The five most called functions/subroutines in `NESTOR` with `ECM`/bilinear (by `gprof`)

```
  %    cumulative    self
 time    seconds    seconds     name
12.21      13.94     13.94      for_cpstr
 9.89      25.23     11.29      intbil_
 9.46      36.03     10.80      read
 7.51      44.60      8.57      close
 6.72      52.27      7.67      __svml_logf4_e9
```

Example 2.3 now provides the same kind of ranking as in Examples 2.1 and 2.2 for the `NESTOR` run involving the `ECM` input domain and bilinear interpolation. For reminders, this particular experiment was the shortest listed in Table 2.1. Watching the console output of `NESTOR` unfold in this case suggests this is because `NESTOR` runs faster while computing vertical levels. Reviewing the source code of `NESTOR` [22] reveals this is indeed due to an optimization of `ECMvgd`, a subroutine whose purpose is to prepare the vertical levels of an input domain using `ECM` as data source. Without going to much into the details here (for those details, see Sec. 2.3.3), this subroutine is more efficient than its peers `ECPvgd` and `NCPvgd` (doing the same job but for `ECP` and `NCP` data sources, respectively) thanks to a `logical` variable maintained through all of its calls (via the `save` instruction of Fortran) which allows `ECMvgd` to avoid redundant (and costly) I/O operations. Thanks to this optimization, Example 2.3 provides a completely different ranking with respects to previous rankings, which can now be explained by the fact that the subroutines `ECPvgd` and `NCPvgd` perform unnecessary I/O operations with respects to `ECMvgd`. In particular, the first entry (`for_cpstr`, a Fortran function for comparing strings) takes only around one eighth of the total execution time, while the first entry in both previous rankings took almost half of the entire run time. The `NESTOR` subroutine `INTbil` (bilinear interpolation) follows closely with almost 10% of the execution time. Nevertheless, because this particular experiment still took almost two minutes, there may still be room for improvement. A first obvious candidate for optimization would be the bilinear interpolation itself, as will be explored in Sec. 2.3.5, but the third and fourth entries suggest that there may also be some potential further improvements in terms of I/O operations.

The first entry in the ranking provided by `gprof` for the `ECM` sample run in Example 2.3 is trickier to analyze. It is indeed strange that a function for comparing strings in Fortran account for so much of the total execution time, as the main tasks of `NESTOR` do not seem to involve string management. To further the analysis of the top entry of Example 2.3, `gprof` was re-run on the `ECM`/bilinear experiment with the `--graph` option in order to display the complete call graph of `NESTOR`. Unfortunately, it listed `for_cpstr` as a *spontaneous* call, meaning `gprof` could not trace back where it was called, most probably because it belongs to an underlying library [4]. To elucidate the role of `for_cpstr`, additional manual testing of `NESTOR` has been performed by removing specific function/subroutine calls or sets of instructions within its source code and running `NESTOR` again. This manual testing eventually showed that `for_cpstr` is most likely involved in the aforementioned optimization implemented in the `ECMvgd` subroutine. Indeed, removing the `logical` variable used by `ECMvgd` to avoid redundant I/O operations (as well as all related instructions) not only slows down `NESTOR` drastically (around 10 minutes of `user` time to prepare 2 days of `MAR` input data due to the large grid dimensions; cf. Table 2.1) but also cuts the total number of seconds spent with `for_cpstr` to only around 2 seconds and an half, which may account for side operations that will not be further explored here due to the difficulty of tracking `for_cpstr` calls. As will be discussed in Sec. 2.3.5, a similar optimization will eventually be presented in Chapter 3 but with a more elegant implementation.

Listing 2.4: The five most called functions/subroutines in `NESTOR` with `MAR`/bilinear (by `gprof`)

```
 %    cumulative   self
time    seconds   seconds    name
28.60     38.45    38.45     intnrg2_
 9.25     50.88    12.43     __libm_cosf_e7
 9.19     63.24    12.36     __libm_sinf_e7
 7.04     72.70     9.46     __libm_atan2f_e7
 5.09     79.54     6.85     intlin_
```

Example 2.4 now provides the usual ranking for the `MAR` on `MAR` forcing scenario. Again, the ranking is completely different from Examples 2.1 and 2.2, but this time around, the most called function/subroutine is the `NESTOR` subroutine `INTnrg2`, which has been specifically designed for forcing the `MAR` model with existing `MAR` output. This is, again, the consequence of some subroutines used by `NESTOR` in this scenario being already optimized: in particular, `MARvgd`, which performs the same operation (i.e., setting vertical levels of the input domain) as `ECMvgd` but for `MAR` grids,

[4] Some additional search [17] seems to point to `for_cpstr` being an Intel function that is added upon compilation by the `ifort` Fortran compiler. However, no detailed, official documentation for `for_cpstr` has been found.

uses the same optimization as ECMvgd to prevent redundant I/O operations. Moreover, INTnrg2 has also been optimized with a comparable mechanism, except that it involves side arrays maintained between each call via the common instruction of Fortran. As a consequence, the MAR sample run is the second shortest experiment in Table 2.1. The functions from the second to the fourth place are all trigonometric functions that are used by the dist function from INTnrg2.f, itself used in INTnrg2, the fifth place being the linear interpolation routine INTlin (found in INTERp.f and used for vertical interpolation). It should be reminded that this INTnrg2 subroutine essentially performs a bilinear interpolation, like INTbil found in INTERp.f, but is slightly more complex due to MAR grids being non-regular (as defined in Sec. 2.1). Interestingly, because the aforementioned subroutine MARvgd also uses a saved logical variable to avoid redundant I/O operations, for_cpstr would be the sixth entry (with 5.55 *self* seconds) in a more complete ranking. The fact that NESTOR spends twice less time on for_cpstr than in Example 2.3 may be due to the difference in dimensions between ECM and MAR grids (cf. Table 2.1), since both ECMvgd and MARvgd are used to prepare vertical levels.

Up to this point, only use cases of NESTOR involving bilinear interpolation have been investigated, which means three more sample runs with bicubic interpolation (as there were seven sample runs, cf. Sec. 2.3.1) needs to be profiled with gprof to have an exhaustive analysis. However, all three cases are pretty straightforward to analyze: in all rankings produced by gprof, the INTbic subroutine from INTERp.f ends up being the top entry. Examples 2.5 and 2.6 provide the rankings for the NCP/bicubic and ECP/bicubic sample runs, respectively. Just like with Examples 2.1 and 2.2, the rankings are essentially identical when it comes to the functions, though the numbers differ because of the differing grid dimensions (cf. Table 2.1). In fact, the rankings are identical to those of the experiments with bilinear interpolation aside the fact that INTbic became the most expensive subroutine.

Listing 2.5: The five most called functions/subroutines in NESTOR with NCP/bicubic (by gprof)

```
  %    cumulative    self
 time    seconds    seconds      name
 43.70     519.63    519.63      intbic_
 28.55     859.09    339.46      netcdf_mp_nf90_get_var_3d_fourbytereal_
  3.10     895.93     36.84      nf_inq_var_
  2.65     927.45     31.52      nf_inq_dim_
  1.74     948.19     20.74      utf8proc_decompose_char
```

The ranking obtained for the ECM/bicubic sample run, shown in Example 2.7, also demonstrates that the INTbic subroutine now constitutes most of the execution time. In fact, in this specific instance, it even constitutes more than 90% of the total execution time. Such a large difference with previous rankings can be explained in two ways. First, as discussed previously, the ECM scenarios already benefit from an optimization in ECMvgd, hence why NESTOR wastes much less time performing I/O operations in Example 2.7 and why the previous rankings were very similar to the use cases involving bilinear interpolation.

Listing 2.6: The five most called functions/subroutines in NESTOR with NCP/bicubic (by gprof)

```
  %     cumulative     self
 time     seconds     seconds      name
39.36      935.46     935.46      intbic_
31.53     1684.79     749.33      netcdf_mp_nf90_get_var_3d_fourbytereal_
 3.30     1763.12      78.33      nf_inq_var_
 2.96     1833.38      70.26      nf_inq_dim_
 1.87     1877.77      44.39      utf8proc_decompose_char
```

Listing 2.7: The five most called functions/subroutines in NESTOR with ECM/bicubic (by gprof)

```
  %     cumulative     self
 time     seconds     seconds      name
94.21     2443.00    2443.00      intbic_
 0.85     2465.14      22.14      read
 0.81     2486.21      21.07      for_cpstr
 0.43     2497.28      11.07      _intel_fast_memcmp
 0.40     2507.69      10.41      close
```

Second, a close inspection of the code of the INTbic subroutine (from INTERp.f) suggests its approach to the problem is very inefficient. For reminders, bicubic interpolation [33] is achieved by computing cubic splines [35] on the rows of the (sub-)grid whose cells are in the neighborhood of the location closest to the interpolation cell, computing one additional cubic spline on the interpolated values obtained by the previous splines, and finally computing the overall interpolated value with this last spline. In theory, it is possible to achieve bicubic interpolation by building splines in a 4 by 4 ***sampling square*** whose inner cells are in the neighborhood of the selected location [27, 33], but the implementation of INTbic actually computes splines over the entire input grid it has to downscale. Even though this implementation reuses efficient code from a reference book on Fortran [27], this makes the bicubic interpolation in NESTOR needlessly costly, as will be detailed in Sec. 2.3.5.

This also explains why the ECM/bicubic scenario features the worst total execution time (almost an hour of user time): not only INTbic processes the entire input grid, but the grids from the ECM input domain are also the largest in terms of dimensions as previously shown in Table 2.1. To refresh NESTOR in this regard, computing cubic splines on 4 by 4 sampling squares should be preferred over considering the entire input grid. Keeping track of these squares may also be a way to prevent NESTOR from performing redundant computations between each iteration of the main loop of NESTOR (as defined in Sec. 2.1). This specific question will be detailed further in Sec. 2.3.5 and Chapter 3.

Before going further with the issues of NESTOR, this performance analysis requires an important closing remark: indeed, NESTOR also provides the option of performing interpolation of vertical levels with either linear interpolation or cubic spline interpolation, as quickly mentioned in Chapter 1. However, such an option has not been discussed at all in this chapter. The reason for this is simple:

in January 2022's NESTOR, using the cubic spline interpolation to prepare air columns above each surface cell of a MAR grid instead of the default linear interpolation (performed with INTlin from INTERp.f) results in bogus data, i.e., the output files are filled with "*not a number*" (NaN) values. This issue will be discussed further in this chapter, in Sec. 2.4.2.

2.3.3 Issue n°1: setting the vertical levels of the input domain

A very obvious issue that arises from the gprof rankings discussed in Sec. 2.3.2 is how NESTOR manages the vertical levels of the input domain. In its unmodified source code (i.e., as of January 2022), this operation is managed by 6 different subroutines (in alphabetical order): CM3vgd, ECMvgd, ECPvgd, LMDvgd, MARvgd and NCPvgd. All six subroutines are called in practice by the VERgrd subroutine, which itself receives the type of large scale input domain (LSC in NESTOR terminology, for **L**arge **SC**ale) and calls one of the aforementioned subroutines depending of this type. The VERgrd subroutine itself is called on a few occasions in the code of the NSTint subroutine. For reminders (see Sec. 2.1), the NSTint subroutine amounts to the main algorithmical operations carried out by a single iteration of the main loop of NESTOR.

Listing 2.8: Example of a call to the VERgrd subroutine in NSTint.f

```
      DO j=1,nj
      DO i=1,ni

        LSC1sp=LSC_sp(i,j)
        LSC1sh=LSC_sh(i,j)

        CALL VERgrd (LSCmod,emptyC,fID,k,nk,LSC1sp,
     .               LSC1sh,LSC1_t,LSC1_p,LSC1hp,
     .               LSCgdz,WK1_1D,WK2_1D,WK3_1D)

        LSC__p(i,j)=LSC1_p(k)

      ENDDO
      ENDDO
```

An important feature of VERgrd is that it is reponsible, at the same time, for building the vertical levels of the input domain **and** the output domain. A logical parameter (i.e., the equivalent of a boolean value in Fortran) tells the subroutine whether it is called to prepare the vertical levels of the input or the output domain. Moreover, VERgrd also receives, as parameters, the pressure, temperature and elevation of a single surface cell, and subsequently computes the pressure levels on top of this surface cell and only this cell. As a consequence, VERgrd must be called for every cell of a grid whose vertical levels must be computed. Example 2.8 shows a typical instance of NSTint calling VERgrd to process the input domain, comments being removed for readability. In this set of instructions, `ni` and `nj` gives the dimensions of a 2D grid (longitude-wise and latitude-wise, respectively) from the input domain while `LSC_sp` and `LSC_sh` corresponds to the 2D arrays giving the surface pressure and elevation of each surface cell of the input domain. The `nk` gives the number of vertical levels, and `k` a specific level to compute. The `LSC1_p` is the vector containing the final vertical levels. Note also the many parameters in the call to `VERgrd`: in particular, the first two parameters give the type of `LSC` (source) data and the `logical` value telling if the subroutine should compute the vertical levels for the input or the output domain. Additional commentary on `VERgrd` will be provided in Sec. 2.4.1.

Listing 2.9: Reading pressure levels in `NCPvgd.f`

```
C +−−−Atmospheric levels: pressure levels
C +   −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
C +        ******
      CALL UNread (fID,'level',0,0,0,0,nk,1,1,NCP__z,
     .             empty1,empty1,var_units,plevel)
C +        ******
```

To elucidate what makes the computation of the vertical levels so slow, the code of the subroutines dedicated to each type of input domain should be investigated. A simple example would be the `NCPvgd` subroutine, as its source code is less than 100 lines of code, and much less if we omit comments and spacing. In this context, the culprit for the slowness of `NESTOR` in the `NCP` sample runs (see Table 2.1 as well as Examples 2.1 and 2.5) is easy to find and shown in Example 2.9 (comments included), which provides the very first instructions of `NCPvgd` after setting the variables. The call to `UNread` (a subroutine from the `libUN.inc` library) reads a specific field (called `level`) of a NetCDF file, provided through the `fID` argument, and stores its values in a `plevel` vector with `NCP__z` receiving the coordinates along the vertical axis.

The main issue with the call to `UNread` in `NCPvgd` shown in Example 2.9 is that it will re-read the `level` variable of the input NetCDF file for every surface cell of the input domain, this variable being a simple one-dimensional vector (this can be easily checked with the `ncdump -h` command). In other words, the content of the `plevel` array should always be the same no matter the surface cell for which vertical levels are being computed. This is especially visible looking back at the double loop shown in Example 2.8. While this code seems innocuous from a semantic point of view, it is catastrophic for performance from a computing science perspective. Indeed, the speed of I/O operations in a computer depends on what type of memory is accessed [28, 29]. The fastest I/O access is typically CPU registers, while the RAM (**R**andom **A**ccess **M**emory) provides a fast access to actively running programs and the data they manage. Reading files on hard drives (such as NetCDF files, in this context), however, is approximately a hundred times slower [28, 29]. This means that a subroutine such as `NCPvgd` not only re-reads the same vector for each surface cell of the input domain, but also does it in one of the slowest possible ways, hence the rankings in Sec. 2.3.2.

As discussed in Sec. 2.3.2, the ECMvgd already fixes this issue by using a single logical variable named lfirst which is carried on throughout the execution of NESTOR via the save instruction of Fortran. This variable is initially set to true and become false after reading the pressure levels vectors it needs to read, knowing the format of ECM data is slightly different. The vectors themselves are also maintained via the save instruction. This means that, after the very first call to ECMvgd, subsequent calls only have to check whether the lfirst variable is set to false to avoid re-reading NetCDF files again. Example 2.10 provides the part of ECMvgd which provides this optimization, cleaned from comments with less spacing. In this code extract, CSTp and SIGp correspond to the vectors that ECMvgd needs to read (but only once) in the NetCDF files. It should be reminded that this approach for speeding up the subroutine is followed identically in MARvgd.

Listing 2.10: Extract from ECMvgd.f showing the optimization

```
      REAL pp,ppm,pps,ppf,pp1,dpsl,hh,empty1(1),CSTp(nk),SIGp(nk),
     .        ECM_sp,ECM__p(nk+1),ECM_hp(nk+1)
      SAVE CSTp, SIGp
      CHARACTER*10 var_units
      LOGICAL lfirst
      SAVE    lfirst
      DATA    lfirst/.true./

      IF (lfirst) THEN

        CALL UNsread (fID,'CSTp',0,0,
     &                0,0,nk,1,1,var_units,CSTp)
        CALL UNsread (fID,'SIGp',0,0,
     &                0,0,nk,1,1,var_units,SIGp)

        DO k=1,nk
          CSTp(k) = CSTp(k) * 1.E-3  !(Pa-->KPa)
        ENDDO

      lfirst = .false.
      ENDIF
```

It should be reminded that, as mentioned in Sec. 2.3.2, the optimization provided in ECMvgd by the unmodified NESTOR has the small apparent downside of making calls to the hidden for_cpstr function more frequent. Nevertheless, Examples 2.9 and 2.10 clearly shows that the subroutines called by VERgrd should all be refactored in a way that ensures they read the vectors they each require only once, or at the very least a minimum of times throughout the entire execution of NESTOR. One way to achieve this would be to include the loop over the 2D grid shown in Example 2.8 within the subroutines themselves. A thorough refactoring of VERgrd and its subroutines will be presented in Sec. 3.2.1, taking also account of subsequent remarks in this chapter (such as in Sec. 2.4).

Finally, on a side note, it is worth mentioning that early NESTOR programmers seemed to have had preferred VERgrd computing vertical levels over a 2D grid rather than a single surface cell, as a comment provided along with the vgd subroutines point out the latter option was picked due to memory constraints. However, comments found in the source code and the documentation provided along with NESTOR [22] do not document this topic further.

2.3.4 Issue n°2: soil texture and soil cover interpolation

While Sec. 2.3.2 focused a lot on vertical levels computation, NESTOR users may also attest of at least two operations carried out at the start of the program that takes some time to complete: the interpolation of soil texture and soil cover over the regional grid, respectively carried out by the subroutines HWSDsl and GLOcov. Indeed, the console output of NESTOR goes as far as printing out the number of the row it finished interpolating during both operations to display their progress to the user. While these operations only occur at the start of NESTOR, they still require a significant amount of time to complete (one or several dozens of seconds with small regional grids). This partially explains why NESTOR still takes around two minutes to complete its execution with use cases like ECM/bilinear and MAR forcing MAR (cf. Sec. 2.3.1).

Listing 2.11: Extract from HWSDsl.f; checking texture in the neighborhood of a location

```
       do k=in-nint(dx1),in+nint(dx1) ; do l=jn-nint(dy1),jn+nint(dy1)
                 kk=k
                 ll=l
        if(kk<1)  kk=cx+kk
        if(ll<1)  ll=cy+ll
        if(kk>cx) kk=kk-cx
        if(ll>cy) ll=ll-cy
        kk=max(1,min(cx,kk))
        ll=max(1,min(cy,ll))

        start(1)=kk
        start(2)=ll
        count(1)=1
        count(2)=1

        Rcode = nf_get_vara_int(NETcid,NET_ID,start,count,cov)
```

Looking into the code of either HWSDsl and GLOcov quickly reveals why these operations are slow: much like VERgrd and the associated subroutines, their interpolation process involves multiple I/O operations. Example 2.11 provides a code extract from the HWSDsl subroutine to discuss the issue, with less spacing than in the source code. This extract provides the beginning of an inner loop within HWSDsl which is meant to explore, for a given location (closest cell to the regional cell to interpolate), the neighboring cells from a large grid providing high resolution texture data. A very similar inner loop also exists in GLOcov, though the subsequent operations differ. In both cases, however, exploring the neighborhood of a given location is meant to make a census of how many cells feature a given texture/cover value, the final value assigned to the interpolated regional cell depending on this census.

In Example 2.11, the dx1 and dy1 give the maximum difference, respectively longitude-wise and latitude-wise, used to explore neighboring cells top, bottom, left or right from a cell that is assumed to be the closest to the regional cell (whose indexes are in and jn). The kk and ll variables are the actual indexes of a neighboring cell, checked by several instructions to ensure they remain within the index ranges of the high resolution grid. Finally, cov is the texture value retrieved by the nf_get_vara_int function, whose return value is a code Rcode provided so the user can document any I/O error [5]. The call to nf_get_vara_int and the values stored in the count tuple (i.e., an array of two values), which provides the size of a sub-grid to load in memory, show that the code retrieves only one pixel from the high resolution grid at once, which is arguably even more inefficient than the I/O operations previously discussed in Sec. 2.3.3.

This peculiar choice of reading data pixel by pixel can however be partially explained. On the one hand, the NetCDF files the HWSDsl and GLOcov subroutines read to get high resolution texture/coverage data are very large: for instance, GLOcov reads a NetCDF file providing a 43200 (longitude-wise) by 21600 (latitude-wise) grid providing texture data for the entire Earth's surface, the file itself weighing around 891 megaoctets. On the other hand, due to how HWSDsl and GLOcov proceed algorithmically, the neighborhood around the in,jn cell will not necessarily constitute a clean rectangular or square area in the high resolution grid due to the varying number of kilometers between two degrees of longitude/latitude, especially when getting close to the poles.

Optimizing HWSDsl and GLOcov will therefore amount, again, to reducing as much as possible the I/O operations. Given that modern day machines now feature dozens of gigaoctets of RAM, it is reasonable to load large chunks of data at once at the cost of loading more pixels than necessary, as this will reduce the slowdown induced by the multiplication of I/O operations on the NetCDF files. Doing so may result in HWSDsl and GLOcov subroutines only taking a few seconds (or less) to complete, and in the I/O calls shown in Examples 2.3 to become negligible. The solution used to achieve this goal will be detailed in Sec. 3.2.2.

[5] In practice, the code of HWSDsl and GLOcov does not check the value of Rcode after calling nf_get_vara_int.

2.3.5 Issue n°3: expensive or redundant interpolation

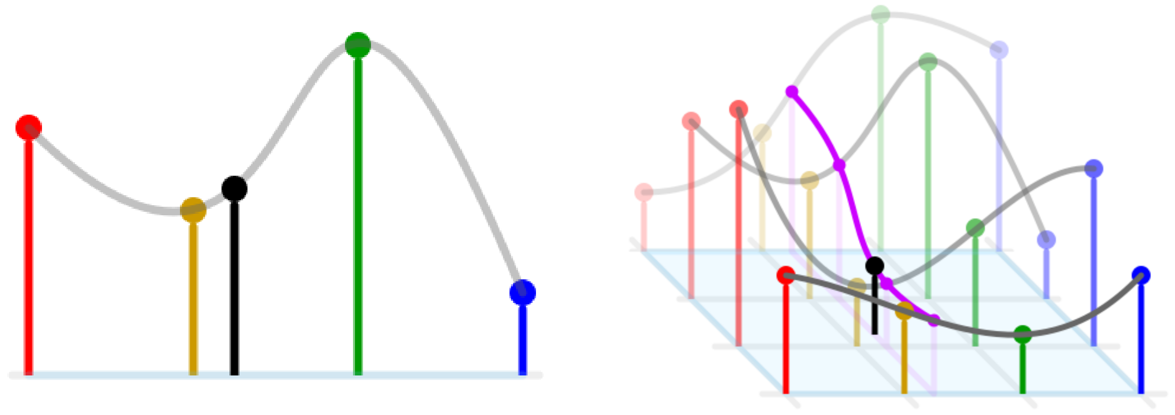
Another clear issue highlighted in Sec. 2.3.1 and Sec. 2.3.2 is the slowness of the bicubic interpolation. In the unmodified NESTOR (i.e., as of early 2022), this task is assumed by the INTbic subroutine found in INTERp.f. The code of such a subroutine is rather straightforward: after adjusting the input values and the arrays providing their positions (longitude-wise and latitude-wise), the code of INTbic can be summarized to Example 2.12 (slightly simplified with respects to comments).

Listing 2.12: Extract from INTbic (in INTERp.f); actual interpolation

```
      ! Construction of 1D splines of rows
      CALL SPLIE2(tmp_Ix,tmp_Iy,tmp_in,dim_Ix,dim_Iy,tmp_I2a)

      ! Interpolation for each output grid point
      DO j=1,dim_Oy
      DO i=1,dim_Ox

        CALL SPLIN2(tmp_Ix,tmp_Iy,tmp_in,tmp_I2a,dim_Ix,dim_Iy,
     .              grd_Ox(i,j),grd_Oy(i,j),var_O(i,j))

      ENDDO
      ENDDO
```

In Example 2.12, dim_Ix (longitude-wise) and dim_Iy (latitude-wise) give the dimensions of the input grid while dim_Oy and dim_Ox provide the same information for the output grid. The tmp_ variables all correspond to pre-processed input data (in particular, tmp_in contains the values of the input grid) with the exception of tmp_I2A, which buffers interpolation data. All four tmp_Ix, tmp_Iy, grd_Ox and grd_Oy provide coordinates of each cell from the input and output grids, respectively. The SPLIE2 and SPLIN2 subroutines are efficient Fortran implementations of numerical methods provided by a 1992 reference book by Press et al. [27]. The purpose of the former is to create a cubic spline for each row of the input grid with the help of another subroutine from Press et al. [27]: the SPLINE subroutine. Prediction of a value with a spline built by SPLINE is done with SPLINT [27]. To understand these subroutines, let us review precisely how bicubic interpolation works.

First of all, a cubic spline is an interpolation method tailored for one dimensional problems: given a sequence of values along an axis, the cubic spline tries to fit the sequence so that new values can be predicted between the known values [35]. As mentioned before, this task is assumed in NESTOR by the SPLINE subroutine. Fig. 2.1a provides a toy example of a cubic spline, built with four values (in red, yellow, green and blue) to predict a fifth value (in black). Bicubic interpolation consists of building cubic splines for each row (or each column) of a (sub-)grid, retrieving interpolated values for the first coordinate of the interpolated point for each spline (predicted in NESTOR with the help of SPLINT [27]), then build a final cubic spline with these values to predict the final interpolated



(a) Toy example of a cubic spline

(b) Toy example of bicubic interpolation

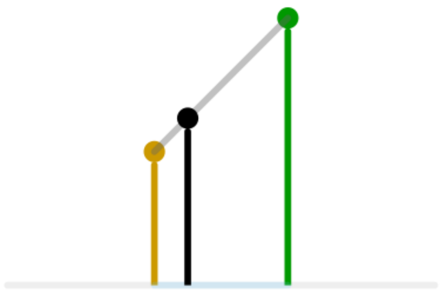
Figure 2.1: Depictions of cubic spline and bicubic interpolation [33]. The black point corresponds each time to the predicted value, while coloured points are known values.

value using the second coordinate (with a combination of `SPLINE` and `SPLINT`) [33]. Fig. 2.1b shows a toy example of bicubic interpolation on a 4 by 4 grid, where red, yellow, green and blue values correspond to individual splines for each row (or column) and where the purple curve represents the final spline used to predict the final value (again, in black). In `NESTOR`, computing the cubic splines for each row of a grid is assumed by `SPLIE2`, as shown in Example 2.12, and predicting the value of each interpolated (regional) cell with the final spline is performed by `SPLIN2`.

As mentioned earlier in Sec. 2.3.2, the main issue of bicubic interpolation in `NESTOR` is the fact that splines are computed over the entire input grid. Let us assume, for instance, that the input grid is a square grid of 120 by 120 cells. This would mean that `INTbic` would first compute 120 cubic splines using 120 values each. Then, for each output grid cell, a new spline must be computed with the 120 interpolated values obtained with each of the first 120 cubic splines. If the computational cost is approximated by the number of values that have to be managed, this means that `INTbic` has to toy with 120^2 (first 120 splines) plus 120^3 values (one additional cubic spline built with 120 values for each of the 120^2 cells), which leads to a total of 1,742,400 values to consider. However, as mentioned in the literature [27, 33], bicubic interpolation can be achieved by using a 4 by 4 sampling square surrounding the location of interest. The same approach as described above can then be followed, giving 4^2 plus 4 values to manage per interpolated value. Multiplied by the dimensions of the grid, this gives a total of 288,000 values to manage, which is only around 16% of the previous total.

Of course, this calculation excludes the cost of finding the sub-grids accomodating the locations associated to each output grid cell. Hopefully, said locations do not change between each iteration of the main loop of `NESTOR` (cf. Sec. 2.1): the longitude/latitude coordinates of both the input and the output domain remain the same for entire execution of `NESTOR`. In other words, it is possible to find the sub-grids in the input domain for each output domain cell only once and re-use the results for each subsequent bicubic interpolation. Such an approach will be developped in Sec. 3.2.3 in order to optimize bicubic interpolation.

In fact, the aforementioned approach for reducing the computational cost of the bicubic interpolation can also be applied to the bilinear interpolation. Indeed, just like bicubic interpolation, bilinear interpolation in NESTOR starts by finding a small set of input grid cells neighboring a given location (as longitude and latitude coordinates). These cells are also called *sampling points* in NESTOR source code. NESTOR subsequently performs a bilinear interpolation [34] with each sampling point and the adjacent cells located bottom, right and bottom-right (for a total of four points). The bilinear interpolation itself consists of performing two linear interpolations [38] between two adjacent cells, using the first coordinate of the location of interest to get interpolated values, then performing a second linear interpolation with these interpolated values and the second coordinate of the location. For reminders, a linear interpolation consists of assuming there is a straight line passing between two points and predicting the value of another point using the slope of this line and the coordinate of this third point. Figure 2.2 depicts linear and bilinear interpolation, respectively in Fig. 2.2a and Fig. 2.2b.


(a) Toy example of linear interpolation

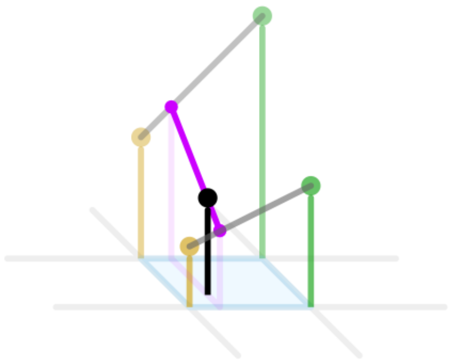

(b) Toy example of bilinear interpolation

Figure 2.2: Depictions of linear and bilinear interpolation [33]. The black point corresponds each time to the predicted value, while coloured points are known values.

As previously implied, the INTbil subroutine of NESTOR (from INTERp.f) actually uses several sampling points from the input grid and their respective adjacent cells to predict the value of a cell in the output grid. It then performs a bilinear interpolation as depicted in Fig. 2.2b between a sampling point and its bottom/right/bottom right neighbors, and assigns, as the final value of the output grid cell, an average of the interpolated values produced with each sampling point.

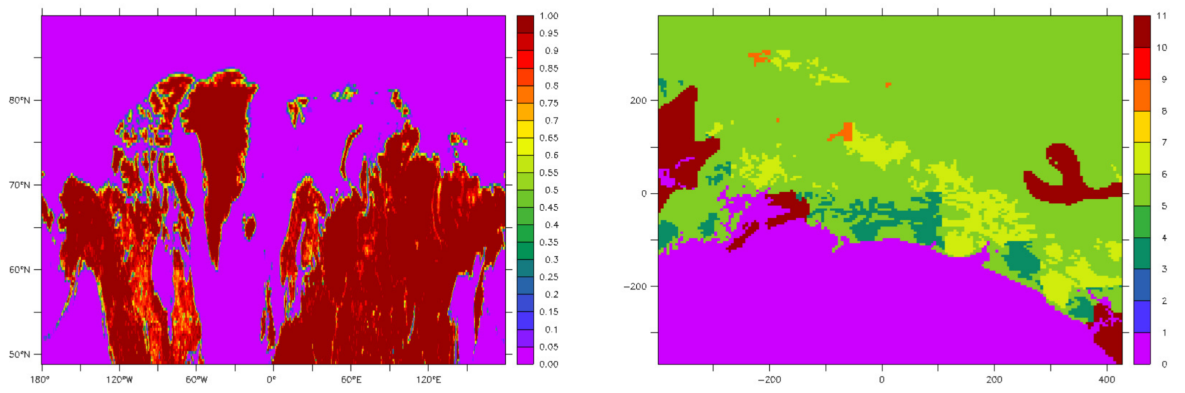
Just like with bicubic interpolation, the longitude/latitude coordinates of each cell in both the output and input grids do not change between each iteration of NESTOR. It is therefore possible to compute only once the sampling points used for each output grid cell during the bilinear interpolation along with their coordinates. By doing so, subsequent interpolations simply consists of applying the linear interpolation formulas with the values of each grid from the input domain, as there is no need to search for sampling points any longer. This is further motivated by the fact that the values from the input domain are used very late in the code of INTbil, meaning most of this code is dedicated to preparing the input coordinates/values and performing sampling point selection [6].

[6]Because INTbil is quite lengthy, no code extract will be provided here.

While the bilinear interpolation is not the most expensive operation performed by NESTOR, going by the rankings presented in Sec. 2.3.2, it can still amount to a significant percentage of the total execution time as evidenced by Example 2.3. Therefore, preventing INTbil from re-doing the sampling search point at each call can prevent redundant computations and reduce the overall execution time. It is worth mentioning, however, that the INTbil is already optimized to some extent: the sampling points selection is already tailored to minimize how many cells from the input domain will be considered. The algorithmic simplicity of INTbil and its few optimizations may explain why modern NESTOR users overwhelmingly prefer the bilinear interpolation (which is also the default interpolation mode). Nevertheless, the INTbil from the early 2022's NESTOR still performs redundant operations if used repeatedly. How to suppress this redundancy will be discussed in Sec. 3.2.3.

2.3.6 Issue n°4: inefficient input domain management

One last issue of NESTOR that has not been discussed in details in Sec. 2.3.2 is the way it manages the input domain. Indeed, the unmodified NESTOR always processes the entire input domain, notably during the computation of its vertical levels (see also Sec. 2.3.3). In practice, this is not necessary: the values NESTOR actually needs lie in a sub-region of the input domain that accommodates the entire regional domain. Even if the full grids from the input domain are available, NESTOR never use points outside this hypothetical sub-region during both interpolation subroutines, especially since said subroutines should select specific cells close to the regional cells to begin with (cf. Sec. 2.3.5).



(a) Large scale grid (latitude band), land/sea mask

(b) Regional grid (gulf of Alaska), soil texture

Figure 2.3: Another use case of NESTOR where the dimensions of the input domain (1200 by 137) induce a slow execution. In both figures, the purple color corresponds to sea and oceans.

At first, this did not appear to be a problem with the experiments discussed in Sec. 2.3.1, though the dimensions of the ECM and ECP input domains explain some user times, especially when compared to the NCP use cases (again, see Table 2.1). However, an experiment performed later with an even larger grid demonstrated that NESTOR was still performing unnecessary calculations and losing time, even after having received solutions to all three issues discussed in Sec. 2.3.3, Sec. 2.3.4 and Sec. 2.3.5. The problematic grid was an ECM grid which covered an entire latitude band in the north of the Northern hemisphere, therefore covering regions such as Greenland, Scandinavia, Siberia, Alaska and Canada. Its dimensions were 1200 cells longitude-wise and 137 latitude-wise. However, the `NESTOR` scenario were such a grid was used aimed at producing a regional grid over a portion of Northern America facing the gulf of Alaska, meaning a very large number of cells from the input domain were useless to prepare the regional data. Despite involving the already optimized `ECMvgd`, this use case of `NESTOR` still took a bit less than four minutes of `user` time to process one month of data, with an already optimized `NESTOR` fixing all previous issues barely dividing this time by a factor of two. For illustration purpose, Figure 2.3 shows the large scale and regional grids of this specific scenario, with Fig. 2.3a showing the land/sea mask of the input data and Fig. 2.3b depicting the soil texture of the output regional data.

A very simple solution to this issue would consist of truncating the NetCDF files tied to the input domain in order to restrict the data to a range of longitudes where the regional grid entirely fits. However, expecting `NESTOR` users to add this additional truncation step to their pipeline (i.e., before even using `NESTOR`) would not be user-friendly: on the contrary, input domain truncation should be another job for `NESTOR`. It should therefore also be tuned such that it does not consider the entire input domain at given steps, such as the computation of vertical levels which is typically performed over the entire input grids (see also Sec. 2.3.3). A partial solution to this problem will also be introduced in Sec. 3.2.4; it will only be *partial* because making `NESTOR` truncate the entire input domain (e.g., as the first step of `NSTint`) would require considerable modifications to the current code. As such, the solution that will be eventually introduced is only meant to minimize the computational cost of the steps where a very large input domain can significantly slow down `NESTOR`.

2.4 Miscellaneous issues

While the technical issues discussed in Sec. 2.3 are the main focus of this chapter, other issues found in `NESTOR` deserve to be shortly discussed here. Indeed, in addition to purely technical problems such as bad management of I/O operations (cf. Sec. 2.3.3 and Sec. 2.3.4), the source code of `NESTOR` is also not always easy to go through because of its occasionally complex subroutine calls, while some features which appear to have been used or planned at some point in the past have been poorly maintained since or poorly tested. Sec. 2.4.1 will first discuss the former issue, while Sec. 2.4.2 will provide a brief review of features of `NESTOR` that are either not usable in practice, either too old or cumbersome to be kept "as is".

2.4.1 Code readability

While not discussed in details in Sec. 2.3.3 via Example 2.8, the list of parameters needed to call the `VERgrd` subroutine is a prime example, if not the worst example of needlessly complex interface that can be found within `NESTOR`. Indeed, `VERgrd` requires no less than 14 arguments, while in practice, only a subset of them are actually useful for the tasks it is meant to carry out.

One of the main reasons explaining why VERgrd has so many parameters is that it is a multi-purpose subroutine: indeed, it is called to prepare the vertical levels of either the input or the output domain regardless of the format of the data that is received or produced. While preparing vertical levels seems to be simple enough to design a multi-purpose subroutine, a quick look into the code of VERgrd and at its calls in NSTint (see also Sec. 2.1) suggests this approach overloads the list of parameters more than anything. First of all, the two first parameters, the 3-characters-long strings LSCmod and NSTmod (as denoted in VERgrd.f), are in practice used in a mutually exclusive manner: either the call to VERgrd provides a valid LSCmod and an empty string, or it provides an empty string in the place of LSCmod and a valid NSTmod. Second, the calls to the subroutines associated to each type of data source or output data format are much, much simpler than the calls to VERgrd. For instance, ECMvgd requires only five arguments, but since the design of the subroutines means VERgrd should still be used as a kind of gateway to the vgd subroutines, the calling code still has to prepare a list of 14 arguments among which most will be useless. Interestingly, early programmers of NESTOR seem to have been at least partially aware of this design flaw, as a small code extract in NSTint.f [7] points out that the temperature parameter is not used at all. This code extract is provided in Example 2.13.

Listing 2.13: Extract from NSTint commenting on the uselessness of LSC1_t

```
        DO j=1,nj
        DO i=1,ni

         LSC1sp=LSC_sp(i,j)
         LSC1sh=LSC_sh(i,j)
C +          ******
         CALL VERgrd (LSCmod,emptyC,fID,k,nk,LSC1sp,
     .                 LSC1sh,LSC1_t,LSC1_p,LSC1hp,
     .                 LSCgdz,WK1_1D,WK2_1D,WK3_1D)
C +          ******
C        LSC1_t ne semble ni affecte ni utilise ! Supprimer ?
```

Another issue lies in the last three parameters WK1_1D, WK2_1D and WK3_1D. These parameters are supposedly arrays that should receive data about vertical coordinates, according to a comment of VERgrd and their use in ECPvgd and NCPvgd. Unfortunately, the documentation of the VERgrd subroutine (and its associated subroutines) does not describe in details what they are used for in practice, as these parameters are not only not taken account of by several subroutines (e.g. ECMvgd and CM3vgd) or only partially required (ECPvgd and NCPvgd require WK1_1D), but they also seem to have no practical use. Indeed, all three arrays are local arrays in NSTint (meaning they are not used elsewhere in NESTOR), and their few occurrences in NSTint only consist of feeding them to VERgrd or (re)fill them with zeros. It is possible that the WK1_1D parameter of ECPvgd and NCPvgd was

[7] Starting at line 1274 in January 2022's NESTOR.

envisioned at some point as a way to avoid the cumbersome I/O operations discussed in Sec. 2.3.3, but their respective code does not take advantage of this possibility. The differences between `WK1_1D`, `WK2_1D` and `WK3_1D` are also not documented, so it is hard to guess what they were supposed to be used for. Perhaps these three arrays were meant to receive each a different format of pressure levels.

All in all, the `VERgrd` is a puzzling subroutine at first glance, especially due to the lack of documentation surrounding some of its parameters. It is possible early `NESTOR` programmers included more parameters than necessary at first in order to add other kinds of algorithms in ulterior versions. The same kind of remark could also be applied to some interpolation subroutines: for instance, `INTbic` requires two temporary arrays as for its two first parameters, said arrays being declared in the `INThor` subroutine (which kickstarts 2D interpolation, calling another subroutine depending on the parameters). However, `INThor` makes no use of these temporary arrays (which are also local to `INThor`), therefore questioning why they are required by `INTbic` in the first place.

A less puzzling issue, but which calls for some simplification, lies in the numerous similarities between `vgd` subroutines. In particular, the only differences between `ECMvgd` and `CM3vgd` [8] are indexes used in the `UNsread` calls. Given how few arguments both these subroutines require, it may be preferrable to go the other way around in this specific instance and add a simple `logical` argument to adjust the `UNsread` calls when necessary, such that there remains only one subroutine (and the associated file) for both use cases.

All these small issues have hopefully no significant impact on the performance of `NESTOR` (if not none at all), but they worsen the readability of the source code for future programmers. This is why Chapter 3 will also discuss ways to simplify a few subroutine calls. In particular, it will discuss how the improved `vgd` subroutines can be made clearer (i.e., their purpose in `NESTOR` is easier to grasp) to future programmers while reducing the number of source files they involve (see Sec. 3.2.1).

2.4.2 Obsolete or unusable features

As hinted multiple times throughout this chapter, January 2022's `NESTOR` also suffers from having multiple features that are, in practice, impossible to use. A simple instance of this problem lies in the option allowing users to use cubic splines for interpolating vertical levels between the input and the output grid, which consists in practice of `NSTint` successively calling the `SPLINE` and `SPLINT` subroutines [27] (see also Sec. 2.3.5) rather than calling the `INTlin` subroutine (linear interpolation). However, as mentioned at the end of Sec. 2.3.2, tuning the `NESTOR` parameters to use cubic splines for vertical interpolation result in the output data being filled with "*not a number*" values, typically displayed in console output with the symbol `NaN`. Fellow programmers know by experience that such an error usually hints the program attempted a division by zero at some point.

Hopefully, finding and fixing this issue is not very difficult. `NESTOR` usually adds one additional level to vertical columns, which is usually set to zero. However, the level before is also typically equal to zero as a result of the computations in the `vgd` subroutines. Having two consecutive zeros in

[8] Found in a `CM3vhd.f` file, which constitutes a typo in the file name.

SPLINE resulted in this subroutine dividing by zero at some point. The solution to this issue simply consisted of modifying the calls to SPLINE in NSTint in order to ignore the additional level. After this modification, vertical interpolation by cubic splines in NESTOR worked as intended. This specific problem however raises the question of whether or not all features available in NESTOR were properly tested by the first programmers.

Regardless of the extent to which NESTOR has been tested in the past, some of its options cannot be used in a meaningful way. In particular, the two *modes* that act as alternatives to the typical downscaling, respectively rain disagregation and wind gust estimation, are unusable. Running NESTOR with the former mode leads to the program stopping at the start of the main loop because it cannot find a specific variable within the input NetCDF files [9], while the latter appears to have been removed: as mentioned in Sec. 2.2, it requires calling a WGustE subroutine that is not present in any source file of early 2022's NESTOR and whose sole call in NESTOR.f has been commented. The reason why the rain disagregation mode does not work may be due to a lack of maintenance over time, as the associated subroutines PRCdes and PRCout seem to be as thorough as NSTint and MARout (for instance).

Just like with previous issues discussed in this chapter (see also the case of the parameters of VERgrd in Sec. 2.4.1), the lack of documentation for the aforementioned modes (and their dedicated subroutines) makes it difficult to know how they can be fixed. It would be therefore preferable to remove such modes and their subroutines, especially since modern NESTOR users do not use them anymore. Another good reason for removing these subroutines is that the way they call the interpolation subroutines from INTERp.f slightly differ from how they are used during the usual downscaling (in NSTing), which can complexify the design of improved interpolation subroutines (notably regarding the required parameters). This is why the improved NESTOR introduced in Chapter 3 will remove these subroutines altogether both to ease the refactoring of the interpolation library and to alleviate the source code of NESTOR for future programmers.

Besides features crippled by a lack of testing or maintenance, the source code of NESTOR also comprises obsolete subroutines or options. For instance, as no one uses LMD data (***L**aboratoire de **M**étéorologie **D**ynamique*) anymore, a few instructions across NSTint and even some variables in NSTdim.inc have no practical use any longer. The same goes for the initialization of NESTOR with sounding data, this possibility also lacking documentation to inform modern users as to how it can be used. NESTOR also contain subroutines that are simply not called anymore, or which appear to be alternative or deprecated implementations of existing subroutines: for instance, MARoutp appears to be an older version of MARout, as it is not called in practice. Moreover, as already mentioned in Sec. 2.1, the non-regular interpolation subroutine INTnrg is never called in practice: because non-regular interpolation is exclusively performed on MAR grids, the current NESTOR exclusively calls the already optimized INTnrg2 subroutine (specifically tailored for MAR on MAR forcing).

Finally, the removal of some obsolete or poorly maintained features of NESTOR has also repercussions on other subroutines. For instance, the INThor1 subroutine, designed as an alternative to

[9]Tested with ECM, ECP and NCP as data sources.

INThor for the PRCdes subroutine, is obviously no longer needed if PRCdes is removed. The same goes for INThorV, a variant of INThor that is apparently tailored to handle the slightly different format of LMD NetCDF files [10]. Removing these subroutines would not only simplify the source code NESTOR, but would also improve its readability to future programmers.

2.5 Updating NESTOR: a roadmap

Despite being practically mandatory to kickstart a simulation with the MAR model, NESTOR has not had a thorough update since 2004, and is in great need of receiving a more consequent update to at least improve its performance. Indeed, as discussed in Sec. 2.3.1, preparing a single month of MAR forcing fields can take around a quarter of an hour or even more with most use cases, the worst use case envisioned in this chapter lasting almost one hour (i.e., ERA-5 data [9] downscaled with bicubic interpolation). Moreover, the source code of NESTOR as of early 2022 exhibits several issues beyond its performance problems, such as having multiple unused, poorly maintained or obsolete subroutines.

Hopefully, the main weaknesses of NESTOR have been well identified and thoroughly commented in this chapter, leading to the three following suggested changes.

- First of all, to address the performance issues experienced while computing the vertical levels of a grid (see Sec. 2.3.3) and to improve code readability, a complete **refactoring of VERgrd and its associated subroutines** should be done. This refactoring should also allow an **improved management of the input domain** (cf. Sec. 2.3.6), as NESTOR also loses time by preparing the entire vertical levels of large input domains.
- Second, **the interpolation library INTERp.f should be reorganized** in order to clean it from outdated subroutines but also to save useful interpolation data that can be kept between iterations of NESTOR. Indeed, since the longitude/latitude of each cell of both the input and the output domains do not change during the execution of NESTOR, sampling points used by any interpolation subroutine can be saved to avoid searching for them again at ulterior calls. Moreover, a **new bicubic interpolation method** should be written to ensure the bicubic interpolation can take advantage of this idea, as the current INTbic subroutine is unfit for it.
- Last but not least, to save a bit of time at the start of NESTOR, **optimizing I/O operations in the HWSDsl and GLOcov subroutines** should be considered.

All three changes, which will be described in more details in Chapter 3, have the merit of being easily feasible. In other words, it is not necessary to refresh the entirety of NESTOR, but only critical operations, with a bit of cleaning here and there to ensure the readability is also improved for future NESTOR programmers. As will be demonstrated later in Chapter 4, these simple changes already drastically improve the performance of NESTOR in all envisioned use cases.

[10] INThorV requires the njv variable from NSTdim.inc, which would be equal to $nj-1$ with LMD data (or similar). In all other cases, i.e. all modern use cases of NESTOR, njv is equal to nj.

CHAPTER 3

IMPROVEMENTS BROUGHT TO NESTOR

This chapter presents the improvements that have been brought to NESTOR in order to tackle the issues described in Chapter 2, and therefore constitutes an answer to question 2 from Chapter 1. First of all, Sec. 3.1 discusses to which extent and how NESTOR has been modified in general and provides a few practical details regarding the new version (e.g., where it can be found). Second, Sec. 3.2 presents the practical changes that have been brought to NESTOR in order to tackle the four main issues discussed in Sec. 2.3 from Chapter 2. Finally, Sec. 3.3 concludes this chapter by discussing how NESTOR could be further improved in the future.

3.1 Methodology to update NESTOR and other practical details

Given the large number of source files found in the source code of NESTOR [22] (roughly 70 `.f` files), but also the length of its main files described in Sec. 2.1 [1], re-writting NESTOR from scratch is off the table. Even if many files ultimately provide a single short subroutine (see for instance `NCPvgd.f`), the time that would be required to re-write the entirety of NESTOR would be too ambitious in the context of this work. This is why this master thesis will rather introduce a few meaningful changes that already improve the performance of NESTOR, as the slowness of early 2022's NESTOR is the main problem its users have with it.

Still, bringing changes to NESTOR should be done in a way that ensures the code will remain consistent-looking to future programmers. In other words, modifications or additions to the current source code of NESTOR should not end up making said source code look like two different programs mangled together. A few simple conventions have therefore been adopted while modifying NESTOR for this master thesis.

[1] `NSTint.f` is more than 2000 lines of code, `INTERp.f` sligthly more than 1500 lines.

- Despite being outdated, the old coding style conventions of Fortran [37] that are still applied by the source code of NESTOR have been kept. This means, among others, that each line of code does not exceed 72 characters and that subroutine names stick to the convention of being maximum 6 characters long [2] to more easily follow the previous convention.
- Whenever possible, the new code of NESTOR calls existing subroutines and functions, either from NESTOR itself or from the included libraries (namely `NetCDF.inc` and `libUN.inc`).
- Likewise, no new external library has been included.
- For convenience, new variables or arrays that had to be kept throughout the entire execution of NESTOR, and in particular sampling points used for interpolation (cf. Sec. 2.3.5), have been declared in `.inc` files. The new `.inc` files follow the same instruction ordering and presentation as existing `.inc` files, such as `LSCvar.inc`, which maintains variables and arrays associated to the input domain for the whole execution of NESTOR.
- For the sake of compatibility with MAR scripts, the format of the control files used by NESTOR (`NSTing.ctr` and `MARgrd.ctr`) has been left untouched.

This does not mean that the new NESTOR has not taken any liberty.

- Thanks to the flexible compilation script (`Compile.exe`) provided along the source code of NESTOR, the refreshed NESTOR includes several new or renamed subroutines. Obsolete subroutines and side associated files (including a few `.inc` files) have been removed altogether.
- Any subroutine declared and implemented in the modified source files that was not called anywhere in NESTOR in practice, either in the unedited source code or following an ulterior modification, has been removed.
- Any component of the new NESTOR that involves interlinked subroutines, in particular the updated interpolation subroutines, has been reorganized into a small library. Each such library groups subroutines based on their purpose, e.g., subroutines involved in bilinear interpolation are provided by the same source file. This convention is essentially only followed by the refactoring of the interpolation library (cf. Sec. 3.2.3), the initial `INTERp.f` lacking readability.

Careful readers will point out that removing unused or obsolete features sounds incompatible with keeping the same control files as before. In particular, `NSTing.ctr` files still reference features of NESTOR that have been previously deemed outdated (see Sec. 2.4.2). To find a middle ground between these seemingly opposite conventions, the main file of NESTOR, i.e. `NESTOR.f`, has been edited throughout this master thesis to enforce certain parameters or to advertise users (via console output) that the (obsolete) features they attempted to turn on have been removed.

[2] In practice, a few subroutines in the unmodified slightly break this convention (e.g., INTnrg2).

On a side note, another change brought by the new NESTOR is a minimized console output: for each iteration of the main loop (cf. Sec. 2.1), the updated NESTOR will only print the path to the NetCDF file being used as well as the time step it is reading, complete with the associated date and time. This modification was motivated by the fact that NESTOR has become fast enough (cf. Chapter 4) for the old console output to be deemed as unnecessary. For legacy reasons, however, a new verbosity flag was added to `NESTOR.inc` to allow one to force NESTOR to print the old console output. This parameter can only be used by editing the source code at the time of writing, but an ulterior update of control files should allow users to turn on and off this verbose mode via said control files.

At the time of writing, the updated NESTOR can be found on the `climato.be` server in the `/srv5_tmp2/jfgrailet/TFE_NESTOR/` folder. Note that this folder is actually the root folder of NESTOR (providing, among others, the `Compile.exe` script, the `NSTing.ctr` control file, the executable `NESTOR.exe` when available, etc.), which means the actual source files will be found in a `src/` subfolder. The updated NESTOR will eventually be available within the MAR gitlab repository [21, 22].

3.2 Practical modifications of NESTOR

Now that the *spirit* of the modifications brought to NESTOR for this master thesis is established, the changes themselves can be described one by one. First, Sec. 3.2.1 presents the refactoring of the `VERgrd` and the `vgd` subroutines which were both slowing down NESTOR and impairing the readability of the source code, as detailed in Sec. 2.3.3 and Sec. 2.4.1 back in Chapter 2. Sec. 3.2.2 subsequently presents a very simple change to speed up the `HWSDsl` and `GLOcov` subroutines that slow down the initial steps of NESTOR, as discussed in Sec. 2.3.4. Sec. 3.2.3 then enumerates the changes brought to the interpolation library (`INTERp.f`), including an entirely new bicubic interpolation subroutine, to reduce the cost of interpolation throughout the execution of NESTOR, therefore addressing the problems pointed out in Sec. 2.3.5. Finally, to partially answer to Sec. 2.3.6, Sec. 3.2.4 introduces the notion of ***input domain truncation***, i.e., the process of reading only a sub-region of the input domain accommodating the output domain to prevent NESTOR from performing useless computations, and apply this idea to the computation of vertical levels.

3.2.1 Refactoring of `VERgrd` and the `vgd` subroutines

As evidenced in Sec. 2.3.2 and Sec. 2.3.3, one of the main reasons why the unedited NESTOR is slow lies notably in unnecessary I/O calls performed by the `vgd` subroutines processing the input domain (except with the already optimized `ECMvgd` and `MARvgd`, cf. Sec. 2.3.2), themselves called by the `VERgrd` subroutine. Not only the repeated I/O calls to read the same levels vector(s) stored in NetCDF files (said vectors depending on the data source, compare for instance `ECMvgd` and `NCPvgd`) constitute a technical flaw, but the design of the `VERgrd` subroutine can be considered to be flawed as well.

One the one hand, `VERgrd` is designed to prepare the vertical levels of a single surface cell (as shown in Example 2.8). While the comments in the source files suggest this was done to optimize memory use at the time of the creation of NESTOR, this constraint seems obsolete with the multiple gigaoctets of RAM available on modern hardware. Moreover, other subroutines of NESTOR called in NSTint (the main caller of VERgrd in the unedited NESTOR), such as the interpolation subroutines, have no issue processing full 2D grids in a single call, which suggests the same thing could be done with the vgd subroutines for convenience. On the other hand, as discussed in Sec. 2.4.1, the multiple use cases of VERgrd and several unused parameters make this subroutine rather puzzling at first.

This is why the first major change brought to NESTOR consisted of a full refactoring of VERgrd and the associated vgd subroutines. This refactoring brought the following changes.

- First of all, VERgrd has been split into two subroutines LSCvgd and NSTvgd, respectively to prepare the vertical levels of the input (LSC) and the output domain (NST). This change was notably motivated by the fact that both tasks call different subroutines in early 2022's NESTOR, with the notable exception of MARvgd (see MARvgd.f in the original NESTOR) which however requires a logical variable (LSCmar) to differentiate the input from the output domain.
- The LMDvgd subroutine has been removed due to LMD data being outdated.
- ECMvgd now receives an additional logical parameter to adapt its calls to UNsread for GCM data, initially prepared with CM3vgd, as the code of ECMvgd and CM3vgd was otherwise identical.
- Likewise, ECPvgd and NCPvgd have been merged into a new subroutine XCPvgd (*X* replacing both *E* and *N*) due to their similarities. A new logical parameter isNCP allows the calling code to run the (very) few instructions that were unique to the initial NCPvgd.
- Just like VERgrd has been split into two subroutines to avoid the multi-purpose issue, MARvgd has been split into MRLvgd (vertical levels of the input domain) and MRNvgd (vertical levels of the output domain). MRL and MRN respectively stands for MAR LSC and MAR NST.
- As a result of the previous changes, LSCvgd can call ECMvgd, XCPvgd or MRLvgd, while NSTvgd can call MRNvgd, CPLvgd or GRAvgd.
- All vgd subroutines have been modified so they receive a 2D grid of surface cells as input, with dimensions being provided as parameters in the case of LSCvgd (due to a specific use case in NSTint) and via NSTdim.inc as for NSTvgd. As a consequence, all subroutines called by LSCvgd now perform their respective I/O calls only once per 2D grid.
- Because they had no practical use in NSTint, WK1_1D, WK2_1D and WK3_1D have been removed. The temperature and elevation parameters have also been dropped for similar reasons.

Thanks to these changes, the new vgd subroutines feature a more readable code, avoid performing repeated and unnecessary I/O calls at the LSC side (that slow down NESTOR, as shown in Sec. 2.3.3) and also require less parameters than initially: LSCvgd requires 9 parameters while NSTvgd only needs 7 (twice less than the initial VERgrd). It is worth noting, however, that the former actually needs 13 parameters in the final updated NESTOR due to 4 additional parameters being required for input domain truncation, as will be discussed in Sec. 3.2.4.

Regardless of input domain truncation, former calls to VERgrd in NSTint could be simplified by removing some parameters and the loops that initially encompassed them to process 2D grids. Finally, VERgrd was initially also called by the subroutines SOUNDg, SNDweb (initialization with sounding) and PRCdes (used by the rain disagregation mode). However, all three were obsolete (cf. Sec. 2.4.2), so they have been removed rather than updated with respects to the new vgd subroutines.

Note that the refactored vgd subroutines still include CPLvgd and GRAvgd (both updated to receive 2D grids rather an individual surface cell) despite these types of output being seemingly not used at all by modern NESTOR users. If they were eventually removed, the refactored vgd subroutines would consist in only 5 source files (as NSTvgd could be entirely replaced by MRNvgd) versus the 9 files used in the unedited NESTOR. Finally, a minor flaw of this refactoring is that repeated calls to LSCvgd will still imply additional I/O reads (one or two) per call, though the final number of I/O calls will remain negligible compared to the total obtained with VERgrd outside the ECM and MAR use cases.

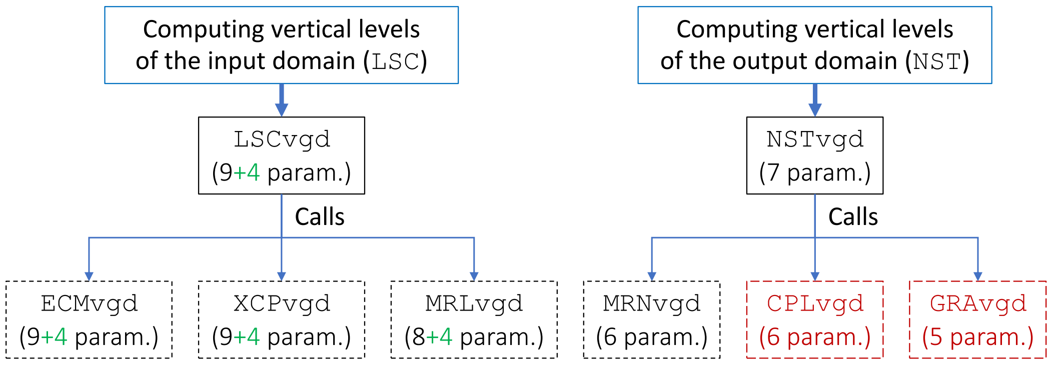


Figure 3.1: Schematic summary of the refactored vgd subroutines in the updated NESTOR. LSCvgd and NSTvgd replace the old VERgrd subroutine, which required 14 parameters.

To conclude this section, Figure 3.1 provides a schematic summary of the refactored vgd subroutines. The green numbers (+4) account for the additionnal parameters required by LSCvgd and its associated subroutines to perform input domain truncation (cf. Sec. 3.2.4) while CPLvgd and GRAvgd are highlighted in red to account for the fact that modern NESTOR users will probably barely use them in practice, or not at all. Finally, while not shown in Figure 3.1, it is important to keep in mind all new vgd subroutines prepare vertical levels for full 2D surface grids rather than for a single surface cell.

3.2.2 Single I/O read of soil texture and soil cover data

The HWSDsl (soil texture) and GLOcov (soil cover) subroutines are two additional examples of subroutines where the unedited NESTOR wastes time because of numerous I/O calls. However, as explained in Sec. 2.3.4, there were good motivations for avoiding loading more data at once: both subroutines are reading high resolution grids stored in files that can weigh almost one gigaoctet. Yet, their reading strategy remains flawed, as both subroutines read one cell at a time, which is slow enough for the console output of the unedited `NESTOR` to advertise their progress.

A simple strategy to solve this issue consists of loading the sub-region of the high resolution grid that fully includes the output domain. In a way, this is the same idea as for the aforementioned input domain truncation (cf. Sec. 3.2.4). Determining this sub-region is not a difficult task: it simply consists of finding the minimum and maximum longitude and latitude coordinates of the output domain, find the corresponding indexes with respects to the dimensions of the high resolution grid, then load the whole sub-region they delimit. Since the algorithms of both `HWSDsl` and `GLOcov` look for the cells that are the closest to regional cells, the sub-region can be slightly extended in all directions in case cells outside the sub-region of the high resolution grid could be valid sampling points.

The only hurdle of this strategy lies in handling the extreme case where the algorithms use cells that are located around 180 degrees of longitude (West or East): in this scenario, the sub-region may be split into two chunks located at both sides of the high resolution grid. It is naturally possible to detect such a scenario, load both chunks then concatenate them together, but this requires additional variables and instructions that are unnecessary in all other cases.

A simple yet effective solution to this issue, which was chosen to save time, consists of loading a latitude band regardless of the longitude coordinates of the sub-region. The rollovers over the boundaries can then be handled exactly like in the original code. While this means loading much more data than necessary, it is an effective strategy: there is indeed a single I/O read garanteeing all cells that could interest `HWSDsl`/`GLOcov` are in memory. A side subroutine `bufLim` (located in its own source file `bufLim.f`) has been written specifically to determine the minimum and maximum indexes along the latitude axis of the high resolution grids to later extract the appropriate latitude band. The outputs of such a subroutine can then be used to load an `allocatable` block that will store the high resolution data. Another benefit of this strategy is that the `HWSDsl` and `GLOcov` subroutines eventually required very few modifications with respects to their initial code.

With this simple change, the interpolation of both soil texture and soil cover is now performed near instantaneously while providing identical output (as will be detailed in Chapter 4), all while avoiding to load the full high resolution grid in memory. This also means displaying the progress in the console output is no longer needed. This change may still be improved in the future by loading the ideal sub-region rather than a complete latitude band in order to minimize memory use, though this memory use is short-lived in practice: once any of both aforementioned subroutines completed their respective tasks, the `allocatable` block used to buffer the high resolution data is freed.

3.2.3 Reorganized interpolation library with new bicubic interpolation

Much like the `VERgrd` and the `vgd` subroutines (see Sec. 3.2.1), the `INTERp.f` library (which provides most interpolation subroutines) needed to be reworked both to improve the performance of `NESTOR` and to improve the readability of its source code. Indeed, the initial `INTERp.f` can be quite intimidating at first sight, as most of the interpolation code of the unedited `NESTOR` is found in this file.

Moreover, there are two variants of the INThor subroutine, which is designed to kickstart horizontal interpolation of a 2D grid and calls either INTbic, INTbil or INTnrg2. These variants (INThorV and INThor1) can be puzzling too, as they exhibit very few differences with the original INThor.

Hopefully, INThorV and INThor1 are only called in situations tied to obsolete features (cf. Sec. 2.4.2). Assuming the rain disagregation subroutines have been removed from NESTOR (as announced in Sec. 3.2.1), a first simplification of INTERp.f therefore consists of removing the INThor1 subroutine as well as the INThorV and the associated call in NSTint. Indeed, INThorV is only relevant for the case where the variable njv from NSTdim.inc differs from nj, which is in theory only true with LMD data. As already reminded in Sec. 3.2.1, today's NESTOR users no longer use LMD data, so all occurrences of njv can be replaced with nj and all sets of instructions related to LMD data can be removed as well. This not only simplifies INTERp.f, but also NSTint.f.

Of course, the previous removals are only improving code readability for future programmers. Actual optimization of NESTOR regarding interpolation methods require more significant changes. First, as already discussed in Sec. 2.3.5, a key strategy to optimize NESTOR in this regard consists of exploiting the fact that longitude/latitude coordinates of cells in both the input and the output domain will not change between each iteration of the main loop in NESTOR.f. Therefore, the sampling points and sampling squares (in the case of bicubic interpolation [33]) used by the interpolation subroutines should be identical at all iterations. As such, the sampling points/squares computed during the very first call to an interpolation subroutine can be saved and reused at ulterior calls, such that these calls can skip the search for these sampling points/squares.

Before reorganizing the code of INTERp.f, the bicubic interpolation should be thoroughly refactored [3]. Indeed, the INTbic subroutine from January 2022's NESTOR is not suitable for the suggested optimization, since it computes splines over the entire input grid and not over sampling squares as recommended in the literature [27, 33]. A refreshed bicubic interpolation should therefore consists of first establishing the best sampling squares for each regional cell, then re-using the code from INTbic on each sampling square to get the interpolated regional values.

Finding a sampling square in the input grid for each regional location (output domain) can be hopefully easily implemented. The first step consists of looking, for each regional cell, for the cell in the input domain that is the closest to the regional location. This first task can be implemented very quickly by re-using the dist function that already exists in INTnrg2.f, though it can be located anywhere in NESTOR. The second step is trickier, as it consists of adjusting the 4 by 4 sampling square such that the 16 cells are the closest to the regional cell. Indeed, because of the 4 by 4 dimensions, the closest large scale cell cannot just be chosen as the center of this square: at best, it can be in the inner 2 by 2 square. The positioning of this center of the sampling square can be optimized by checking which of the 4 corner cells of the encompassing 5 by 5 square (where the closest large scale cell can be truly in the center) is the closest to the regional location. Based on this result, the sampling square can be adjusted to minimize the distance between each of the 16 cells and the regional location.

[3] Note that this sequence of ideas is used because it fits better the developments of this manuscript, but in reality, optimization of INTbil has been implemented first due to being easier to design, implement and test.

The closest corner of the encompassing 5 by 5 square simply becomes the cell that is diagonally the farthest from the closest large scale cell (i.e., closest to the regional cell).

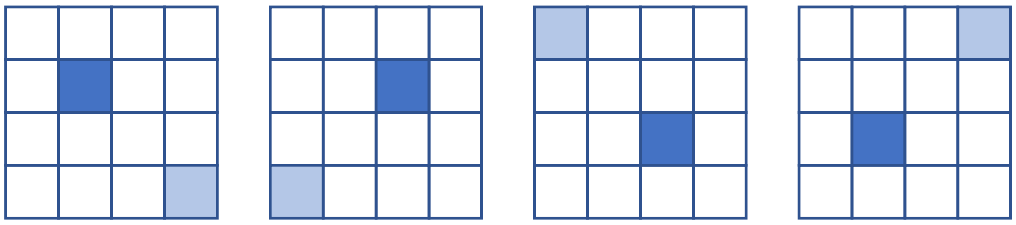


Figure 3.2: Visual representation of how the new bicubic interpolation adjusts a sampling square. The dark blue cells correspond to the cell from the input domain that is closest to the location to interpolate, while light blue cells correspond to the closest corner of the encompassing 5 by 5 square.

Figure 3.2 illustrates the sampling squares the updated `NESTOR` deduces during bicubic interpolation for each regional cell to interpolate. In each of the four squares, the dark blue cell corresponds to the cell from the input domain which is the closest to the regional cell, based on the result of the `dist` function. The light blue cell corresponds to the corner of the encompassing 5 by 5 square which is also the closest to the regional cell. As shown by Figure 3.2, the sampling square is adjusted such that the closest corner is also the cell that is the farthest from the dark blue cell (i.e., large scale cell closest to the regional cell) within the sampling square. This heuristic aims at minimizing the distance between the regional cell and each of the 16 cells of its associated sampling square, such that the resulting interpolation is as accurate as possible. On a side note, it is worth noting a sampling square can be saved by the code by keeping track of the indexes of its top left corner cell.

Once a sampling square has been found and adjusted to each regional cell, the instructions of the original `INTbic` can be re-used on each sampling square, which also means `SPLIE2` and `SPLIN2` [27] will be reused as well. Interestingly, the `SPLIN2` subroutine found in early 2022's `NESTOR` is not actually the code provided by Press et al. in their book [27], but a "*rough optimization*" according to a code comment [4]. Unfortunately, the comments do not document exactly how it was optimized, even though the code looks like it roughly includes parts of `SPLINT` [27]. Due to this lack of documentation, the `SPLIN2` subroutine from the unedited `NESTOR` has been replaced with the version provided by Press et al. [27] to ensure accuracy of the interpolation process.

With slightly less code and a refreshed bicubic interpolation, the interpolation library can be reorganized to prevent redundant sampling data search while improving code readability. Preventing redundant computations is not difficult to implement in practice, as it simply consists of decomposing the bilinear and bicubic interpolation subroutines (knowing the non-regular interpolation used for `MAR` forcing is already optimized, cf. Sec. 2.3.1) into two separate subroutines each, with the first one finding the sampling points/squares and the second one performing the actual interpolation on that basis. The outputs of the former can be saved for the entire execution of `NESTOR`, meaning it

[4]The actual comment is in French: "*version grossièrement optimisée*".

needs to be called only once, while the latter will be called at every interpolation. To avoid overloading INTERp.f with multiple subroutines, the different types of interpolation have been moved in side files (therefore applying one of the guidelines mentioned in Sec. 3.1) with INTERp.f keeping the most general subroutines, such as INThor. More precisely, the following changes have been made.

- The new bilinear interpolation subroutines bilSet (sampling points search) and bilDo (actual interpolation), derived from the initial INTbil, have been moved to a new intBil.f file.
- The same goes for the refreshed bicubic interpolation, i.e., bicSet (establishing sampling squares) and bicDo (actual interpolation) are found in intBic.f.
- The variables and arrays that maintain the sampling points or squares in order to avoid redundant computations during a bilinear or bicubic interpolation are respectively declared in intBil.inc and intBic.inc.
- For consistency's sake, the INTnrg2 subroutine has been renamed intMAR, just like its source file (intMAR.f rather than INTnrg2.f), this name better reflecting its practical purpose.
- Because the dist function is now involved in both bicSet and intMAR, it has been moved in INTERp.f to acknowledge its now more general purpose.
- Conversely, SPLIE2 and SPLIN2 [27] have been moved to intBic.f since they are exclusively used in the context of bicubic interpolation (SPLINE and SPLINT are used in other contexts).

After applying these changes, the initial INThor subroutine can be slightly updated to take advantage of the decomposition of the interpolation subroutines. It simply consists of adding a logical variable, maintained through the execution of NESTOR via the save instruction (just like in Example 2.10 from Sec. 2.3.3), to remember whether or not the bilSet/bicSet subroutines have been already called once. The code of INThor, which was also renamed intHor to keep a consistent naming convention, is otherwise the same as before.

For illustration purpose, Example 3.1 shows the updated instructions of intHor regarding the calls to bilinear and bicubic interpolation subroutines. In this code extract, grd1Ix and grd1Iy provide respectively the longitude and latitude coordinates of each cell from the input domain (assuming it is a regular grid, cf. Sec. 2.1) while grd_Ox and grd_Oy assume the same roles for the output domain. The ifirst variable is the aforementioned logical variable used to know whether or not sampling points/squares are already available [5]. The SPHgrd is an additional logical parameter that is inherited from the unedited NESTOR and still used to signal the use of spherical coordinates in the input domain. Finally, the var_I provides the values from the input domain, and therefore, var_O collects the interpolated values needed on the output domain.

[5]It is worth noting this code also assumes that NESTOR uses bilinear and bicubic interpolation in a mutually exclusive manner. Should an ulterior version use both for some reason, the code would need to be adapted in this regard.

Listing 3.1: Extract from the updated `intHor`

```
! Bilinear interpolation
IF (intype.EQ.1) THEN

  IF (ifirst) THEN
   CALL bilSet(grd1Ix, grd1Iy, SPHgrd, grd_Ox, grd_Oy)
   ifirst=.false.
  ENDIF
  CALL bilDo(grd1Ix, grd1Iy, var_I, SPHgrd, var_O)

! Bicubic interpolation
ELSE IF (intype.EQ.3) THEN

  IF (ifirst) THEN
   CALL bicSet(grd1Ix, grd1Iy, grd_Ox, grd_Oy)
   ifirst=.false.
  ENDIF
  CALL bicDo(grd1Ix, grd1Iy, var_I, grd_Ox, grd_Oy, var_O)

ENDIF
```

Finally, two last minor changes brought to `INTERp.f` deserve a mention. First, the `INTlin` linear interpolation subroutine has been renamed `intLin` to ensure the refreshed interpolation library always uses the same naming convention. Second, a variant of `INTbil`, used in the context of the `ETOPO1` and `ICEmsk` subroutines, has been moved into `INTERp.f` under the new name `bilSim` (`Sim` for "*simple*"). Such a subroutine was initially declared and implemented in `ETOPO1.f` under the name `INTsimple`. While it first looks almost identical to `INTbil`, there are a few subtle differences which prevent using the subroutines from `intBil.f` and `intBic.f` "as is". Hopefully, this variant of bilinear interpolation is called only once in both `ETOPO1` and `ICEmsk`, which are only called at the initial steps of `NESTOR` upon preparing `MAR` input data for a simulation over Antartica. In other words, even when they are used by `NESTOR`, these specific interpolations do not need to be optimized.

3.2.4 Input domain truncation during the computation of vertical levels

The main motivation for reorganizing and partially refactoring the interpolation library of `NESTOR` was to prevent it from performing redundant computations, i.e., in this case, searching again for sampling data at each new call of a 2D interpolation subroutine. Before that, the previous sections (Sec. 3.2.1 and Sec. 3.2.2) described changes designed to minimize the I/O calls that slow down `NESTOR` due to the large difference in memory access speed between reading data stored in the RAM (i.e., program data and instructions) and stored on a disk (like a NetCDF file, in this context) [28, 29]. This section now introduces yet another way to speed up NESTOR: avoiding useless calculations. Because the output domain typically only covers a portion of the input domain, some steps of NESTOR actually perform unnecessary computations. For instance, the preparation of vertical levels is performed on the entire input domain in the unedited NESTOR, despite that NESTOR will only need the vertical levels within the sub-region that encompasses the output domain. Therefore, computing vertical levels outside that sub-region of interest is a waste of time in practice.

The process of restricting operations on the input domain to a sub-region will be referred to as ***input domain truncation***. This idea could be applied to various steps of NESTOR. In a way, the approach used in Sec. 3.2.2 to speed up HWSDsl and GLOcov already partly follows this idea since it restricts the loaded data to a latitude band (as already discussed, this could be optimized further). Moreover, determining an ideal sub-region encompassing the output domain is an easy task: it only requires finding the extreme values of the longitude/latitude coordinates from the output domain then selecting the index ranges in the input domain that guarantee these extremes will be included.

However, exploiting these index ranges throughout the code of NESTOR is a not-so-easy task and requires significant changes across the entire program, notably due to how the source code is built around the idea of knowing in advance the dimensions of the input and output domain via NSTdim.inc. Indeed, if the index ranges of the sub-region are computed by NESTOR itself, then the arrays associated to the truncated domain would have to be dynamically allocated via the allocatable keyword contrary to the arrays associated to both the input and the output domains. Otherwise, the NSTdim.inc file could be expanded to also include the index ranges of the sub-region within the input domain, but this would then require the user to know them in advance or compute them via a side program. Either way requires a significant change of approach with respect to early 2022's NESTOR, while all changes suggested in previous sections were implemented while staying consistent with the initial code and coding style (cf. Sec. 3.1). A third, simpler option would consist of keeping the arrays associated to the input domain and only filling the cells of the sub-region, but this would also mean NESTOR would waste a significant amount of memory.

This is why the idea of input domain truncation has been only applied to the vgd subroutines dedicated to the input domain side (hence the green numbers in Fig. 3.1) in the context of this master thesis. To do so, the following changes have been brought to NESTOR in addition to the previous changes (in particular those described in Sec. 3.2.1).

- NSTint now contains a few more variables and some additional instructions (found in the beginning of the subroutine) to determine the index ranges of the sub-region of the input domain that accommodates the output domain. In particular, baseI and maxI provide the extreme indexes (lowest and highest, respectively) along the longitude axis, while baseJ and maxJ provide the same indexes for the latitude axis.
- To ensure there is some leeway around the sub-region, in case interpolation algorithms would need to look for sampling points/squares close to the boundaries of the truncated domain,

some padding has been added around the sub-region. This padding is 5 cells wide. E.g., a 20 by 20 sub-region would be extended to a 30 by 30 sub-region (5 cells added to each side).

- The `vgd` subroutines designed to prepare vertical levels of the input domain have all been modified to receive the `baseI`, `baseJ`, `maxI`, `maxJ` variables delimiting the truncated domain. These four variables therefore correspond to the +4 annotations shown in Figure 3.1 in Sec. 3.2.1. These four variables are simply used to set the ranges of the loops used in said subroutines, therefore effectively ignoring all surface cells outside of the sub-region.

While this solution could be perfected, notably by finding better ways to share the index ranges across all subroutines and by computing them only once, it has the merit of already drastically speeding up `NESTOR` when using large input domains. After applying the aforementioned changes, the use case discussed in Sec 2.3.6 could be fully processed in less than one minute, versus a bit less than 2 minutes with the updated `NESTOR` without said changes and around 4 minutes with the unedited `NESTOR` (see also Chapter 4). This suggests `NESTOR` could become even faster by generalizing input domain truncation, as other steps can still suffer from considering the entire input domain, such as the search for sampling squares in the `bicSet` subroutine (see also Sec. 3.2.3 and Sec. 4.2.2).

Finally, on a side note, it is important to stress that the aforementioned changes assume that the output domain is always fully included in the input domain, i.e., if the input domain covers the whole longitude range, the output domain will not be overlapping the West/East borders. Hopefully, `NESTOR` users already rely on this assumption, which could also be used to motivate more economic memory allocations in `HWSDsl` and `GLOcov` (cf. Sec. 3.2.2) [6].

3.3 Closing comments: a refreshed NESTOR, yet still improvable

The various changes brought to `NESTOR` described throughout this chapter have the merit of being simple, in the sense that they do not significantly change the architecture of the program nor its calculations, with the obvious exception of the bicubic interpolation (cf. Sec. 3.2.3). Despite being rather light, these changes still address all the issues that were put forward by Chapter 2.

However, this does not mean the updated `NESTOR` could not be improved further. As discussed in Sec. 3.2.4, input domain truncation is still in its infancy at the time of writing. To better apply this idea would require carefully updating the code of multiple subroutines of `NESTOR`, as well as re-evaluating some of its assumptions. In particular, by assuming the full inclusion of the output domain within the limits of the input domain, the memory allocations currently performed in the updated `HWSDsl` and `GLOcov` may be optimized.

Before discussing in details the future of `NESTOR` in Chapter 5, Chapter 4 will first assess both the performance of the new `NESTOR` and the sensitivity of the `MAR` model to the forcing data it provides, notably when produced with the new bicubic interpolation.

[6] It should be noted, for context, that input domain truncation was the last change I brought to `NESTOR` for this master thesis. At the time of updating `HWSDsl` and `GLOcov`, I still considered overlapping input domain borders as a possibility.

Chapter 4

Assessment of the updated NESTOR and sensitivity of the MAR model

This chapter thoroughly assesses the updated NESTOR presented in Chapter 3. Not only this chapter demonstrates how much faster NESTOR has become, but it also aims at evaluating whether or not small decimal changes observed with the new version have an impact on the MAR model, and whether or not the new bicubic interpolation results in different or comparable outputs (and to which extent). Therefore, this chapter provides answers to both questions 3 and 4 from Chapter 1. Sec. 4.1 first provides more context regarding the decimal changes to better justify the evaluation methodology used throughout this chapter. Second, Sec. 4.2 directly compares early 2022's NESTOR with the updated version. Sec. 4.3 subsequently discusses the sensitivity of the MAR model with respects to the updated NESTOR, regardless of the type of interpolation. Finally, Sec. 4.4 summarizes the main results of this chapter before moving to the general conclusion (Chapter 5).

4.1 The multiple ways to evaluate the updated NESTOR

Usually, assessing the correctness of a program that has just received one or several updates can be easily done by comparing the outputs of the updated program with respects to those of a previous version. If these outputs are identical, it can be assumed that the new program still performs the same job as before. Unfortunately, this simple strategy cannot be used in the case of NESTOR because the vast majority of its variables (e.g., temperature, pressure, specific humidity, etc.) are real values.

In a computer, real values are typically stored in floating-point representation with finite precision [15]. This means that, if a real value has an arbitrary long number of decimals, only a certain number of these decimals may be accurately represented. In Fortran standards contemporary of the first NESTOR, real values are typically represented with either *simple* or *double precision*. Real values represented with simple precision have 6 significant digits of precision, i.e., the 6 first decimals should be correct [30, 31]. With *double precision*, the number of significant digits rise to 15 [30, 31]. Having a limited number of significant digits of precision means real values in a computer program are bound to have ***rounding errors*** when each feature a large number of decimals [15].

While two computer programs running in the exact same way (and using no random values) should produce the same outputs with the same inputs, the outputs may vary if one of both programs feature a different ordering in terms of instructions. The rounding errors will differ due to the new ordering and propagate, eventually resulting in decimal changes between the outputs of each program. This was practically observed very early with NESTOR during this master thesis: indeed, one of the first changes ever made in the context of this work was an attempt at decomposing `INTbil` in several subroutines in order to test the idea elaborated in Sec. 3.2.3. However, this meant that the flow of instructions between the newly edited NESTOR and the old version was slightly different (despite using the same algorithms), and therefore, decimal changes quickly appeared.

Early quantification of the decimal changes hopefully showed that the errors between early 2022's NESTOR and the *work in progress* version were actually very small, e.g., the average error for temperature values was of the order of 10^{-5}. However, this is not sufficient to conclude that the updated NESTOR is equivalent to the previous version: indeed, many physical processes (e.g., convection [14, 20, 26]) simulated in computer climate models, including the MAR model, are naturally chaotic [26]. This means that small decimal changes can potentially eventually lead to different results with the same model. In other words, to assess the new NESTOR, the MAR model should be run twice with outputs coming respectively from January 2022's NESTOR (or older) and the updated NESTOR presented in Chapter 3, after what the resulting MAR outputs should be compared to assess whether or not the updated NESTOR eventually led to comparable results. Obtaining different results would suggest the decimal changes are problematic, and conversely, strong correlation between both MAR outputs would demonstrate the updated NESTOR can be safely used.

This is why the updated NESTOR will be evaluated in at least three ways throughout this chapter.

- First of all, it will be compared directly with January 2022's NESTOR in Sec. 4.2. On the one hand, the decimal changes between both versions will be quantified. On the other hand, the performance of the updated NESTOR will be carefully assessed in order to evaluate how much faster it is. For reminders (cf. Chapter 1), improving the performance of NESTOR is one of the main motivations for this master thesis.

- Second, the MAR model has been run with a given regional setting (i.e., simulating the climate over mainland Scotland for an entire year) twice: a first time with forcing fields as produced by early 2022's NESTOR (with bilinear interpolation), and a second time with the same data as prepared by the new NESTOR (also with bilinear interpolation). The outputs of the MAR will be compared to evaluate if the MAR model produced comparable or different results as a consequence of the decimal changes of the updated NESTOR. The precise settings of these experiments and their comparison will be provided respectively in Sec. 4.3.1 and Sec. 4.3.2.

- Finally, using a third MAR simulation run with the same regional setting and period as before, the effects of using forcing fields prepared with the updated bicubic interpolation will be assessed. This final evaluation aims at determining whether or not the bicubic interpolation changes the results of a MAR simulation and to which extent. This will be discussed in Sec. 4.3.3.

4.2 Accuracy and performance of the updated NESTOR

This section assesses the updated NESTOR by directly comparing it to January 2022's version. Sec. 4.2.1 first compares the outputs of the updated NESTOR with those of the old version. This sort of validation aims at demonstrating that the small decimal changes observed with the NESTOR introduced in Chapter 3 are negligible, i.e., small changes in the ordering of the operations did not result in significantly different output files. Sec. 4.2.2 subsequently evaluates the performance of the updated NESTOR on use cases that are comparable to those introduced in Sec. 2.3.1.

4.2.1 Validation of the output files

To validate the outputs of the updated NESTOR, the grids obtained for seven representative variables and for the same region and time period have been compared in five different use cases. Four of these five use cases involved the four different data sources already used in the experiments from Sec. 2.3.1, i.e., there is an ECM scenario, a NCP scenario, etc. The fifth scenario is the one described in Sec. 2.3.6, i.e., the production of forcing fields for a region next to the Gulf of Alaska using an input domain covering an entire latitude band. This scenario was added both for exhausitivity and to have a more ambitious use case to assess the performance of NESTOR in Sec. 4.2.2.

For each use case, both versions of NESTOR have been run with the same region, the same time period, the same duration (30 or 31 days) and the bilinear interpolation. The bicubic interpolation has not been considered here: indeed, the algorithm itself has been modified (cf. Sec. 3.2.3), which means it is very unlikely both versions of NESTOR would produce the (exact) same outputs with bicubic interpolation. For an easier comparison, both versions of NESTOR have been tuned to provide their output data in a NetCDF file in addition to the regular binary files they build for the MAR model.

The seven variables that have been compared are given below.

- UU: longitude-wise wind speed component.
- VV: latitude-wise wind speed component.
- TT: real temperature.
- PT: potential temperature [1].
- QQ: specific humidity.

[1]Temperature the air parcel would have if brought adiabatically to the reference 1000 hPa level [14, 20].

- SP: pressure thickness.
- SST: sea surface temperature.

It is worth noting that, for the first five variables of the list, all levels (24 in total, as recommended for the MAR model) have been compared. For the SP and SST variables, only the surface level is available due to their nature. Finally, the variables regarding the soil texture and cover (TEX and SOL in the NetCDF files) have been omitted: because the values for these variables are treated as simple integers, the updated NESTOR produced the exact same results as the old version.

Data	**Chg**	UU	VV	TT	PT	QQ	SP	SST
ECM	Avg	$5.3*10^{-6}$	$4.9*10^{-6}$	$3.5*10^{-5}$	$3.3*10^{-5}$	$2.5*10^{-4}$	$9*10^{-6}$	$3.7*10^{-5}$
	Max	$7.1*10^{-4}$	$6*10^{-4}$	$3*10^{-4}$	1.03	$1*10^{-3}$	$9.1*10^{-5}$	$1.5*10^{-4}$
ECP	Avg	$1*10^{-5}$	$1*10^{-5}$	$5.9*10^{-5}$	$5.4*10^{-5}$	$2.5*10^{-4}$	$1.5*10^{-5}$	$6.7*10^{-5}$
	Max	$7.4*10^{-4}$	$7.2*10^{-4}$	$2.1*10^{-4}$	$1.8*10^{-4}$	$1.1*10^{-3}$	$9.1*10^{-5}$	$1.8*10^{-4}$
NCP	Avg	$7.1*10^{-6}$	$6.5*10^{-6}$	$3.5*10^{-5}$	$3.2*10^{-5}$	$2.7*10^{-4}$	$1*10^{-5}$	$4*10^{-5}$
	Max	$8.9*10^{-4}$	$6.2*10^{-4}$	$3.2*10^{-3}$	0.68	$1.1*10^{-3}$	$9.1*10^{-5}$	$1.8*10^{-4}$
MAR	Avg	0	0	0	0	0	0	0
	Max	0	0	0	0	0	0	0
GRD	Avg	$4.6*10^{-6}$	$3.9*10^{-6}$	$3.2*10^{-5}$	$3*10^{-5}$	$4*10^{-9}$	$8.4*10^{-6}$	$3.4*10^{-5}$
	Max	$8.7*10^{-4}$	$6.7*10^{-4}$	$3.2*10^{-4}$	0.051	$6.3*10^{-7}$	$9.1*10^{-5}$	$1.8*10^{-4}$

Table 4.1: Decimal changes between the unedited and the updated NESTOR for five comparable use cases (processing of 30 days for the same region and time period with bilinear interpolation).

Table 4.1 provides the results of this validation. The GRD label refers to the aforementioned fifth scenario: the data source is actually ECM to NESTOR (ERA-5 dataset [9]). The GRD label rather denotes the type of region the data covers (GRD is for **Gr**eenlan**d**). For each use case, both the average and the largest decimal changes between both versions of NESTOR are provided. While not shown in the table, it should be noted that the validation program [2] also quantified how many values were not exactly identical between both output NetCDF files. The ratio of output values differing between the old and the new NESTOR ranged from 75% to more than 95% for the selected variables.

The decimal changes shown in Table 4.1 are reassuring, especially when it comes to the average decimal changes: at worst, the average change is of the order of 10^{-4}, with the overwhelming majority of the averages being even lower. Moreover, there was no change at all in the use case of MAR on MAR forcing: this is simply due to the fact that the interpolation subroutine used in this case has not been modified in practice (cf. Sec. 3.2.3). The largest changes are also negligible, with the notable exception of the PT variable (potential temperature) for which the most significant change even reaches the unit with the ECM use case. Hopefully, this large difference does not prevent the average from staying very low. A closer look to the largest decimal changes observed between both NetCDF

[2]Available on climato.be: /srv5_tmp2/jfgrailet/TFE_NESTOR/validation/validation.f90.

files of each use case suggest that they are often located at the levels closest to the surface [3]. This result may be due to the fact that NSTint subroutine processes vertical levels for output variables from the highest pressure level to the surface, causing the decimal changes to become greater as the code gets closer to the surface in some cases. However, this is not a rule of thumb for all variables, as how each variable is processed with respects to vertical levels in NSTint slightly varies.

4.2.2 Performance of the updated NESTOR

Now that the differences that exist between the respective outputs of both versions of NESTOR have been demonstrated to be negligible, a greater question remains: is the updated NESTOR faster than the old one, and to which extent ? To answer this question thoroughly, each interesting use case of NESTOR discussed so far (in both Sec. 2.3.1 and Sec. 4.1) has been run multiple times with the `time` command [5] rather than a single time as in Sec. 2.3.1. There are 9 use cases in total: ECM/bilinear, ECM/bicubic, ECP/bilinear, ECP/bicubic, GRD/bilinear, GRD/bicubic, MAR, NCP/bilinear and NCP/bicubic. 6 of these uses cases (i.e., all except GRD and MAR) produce forcing and surface fields for the same output domain (a portion of Western Europe). The GRD use cases derive from Sec. 2.3.6 (see also Sec. 4.2.1). The MAR use case uses an input domain encompassing British Isles to downscale it to the mainland Scotland, using a bilinear interpolation tailored for non-regular grids.

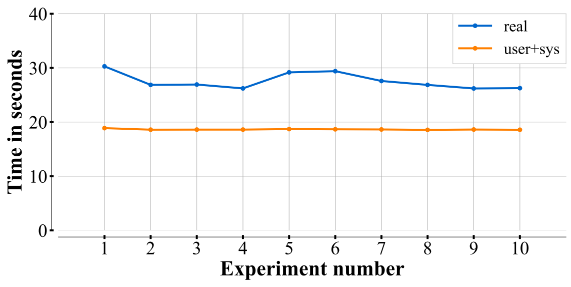
Each use case has been run a total of 10 times. For each experiment, the `real` time and the sum of the `user` and `sys` times have been recorded. For reminders, `real` time gives the actual wall clock time elapsed between the start and the end of the NESTOR run, while `user` gives the time elapsed while NESTOR was actually running on a CPU outside system calls, with `sys` giving the time elapsed during system calls. As a result, the sum of `user` and `sys` times gives a good idea of the time spent by NESTOR on a CPU, especially given that the `sys` time is no longer negligible in some cases.

Running 10 times NESTOR is motivated by the fact that, on an actual shared server such as `srv5` (via `climato.be`), the execution time of any program will fluctuate with the load of said server. When no other computationally intensive program is running, NESTOR can complete a run with a `real` time close to the sum of `user` and `sys`. Otherwise, it may perform slightly worse due to the load. The sum of the `user` and `sys` times should conversely stay almost constant, with slight random variations. Plotting the evolution of both times across all 10 experiments for a given use case gives a realistic outlook on how NESTOR should typically perform [4], while the times themselves can be used to compute mean times that are representative of the performance of NESTOR.

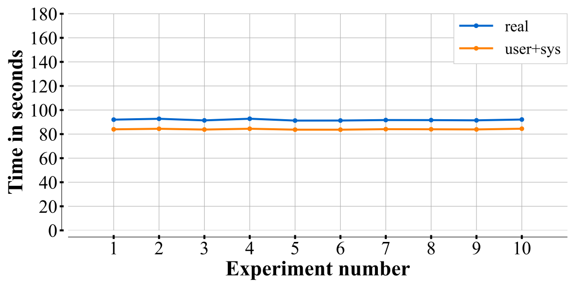
Figure 4.1 provides the performance of the NCP/bilinear (Fig. 4.1a) and NCP/bicubic (Fig. 4.1b) use cases. The former demonstrates a spectacular acceleration with respects to the comparable experiment discussed in Table 2.1: the bilinear case could be completed in less than 30 seconds in all cases, with an ideal time (`user+sys`) even below 20 seconds. In other words, this specific use case

[3]This can be verified by slightly modifying the validation program so that the indexes of the cell with the largest decimal change are kept and eventually displayed.

[4]It is important to note that the repeated runs have not been conducted at selected times, i.e., the load of the server was random. Even more complete experiences would run NESTOR at the same time as, for instance, a running MAR simulation.



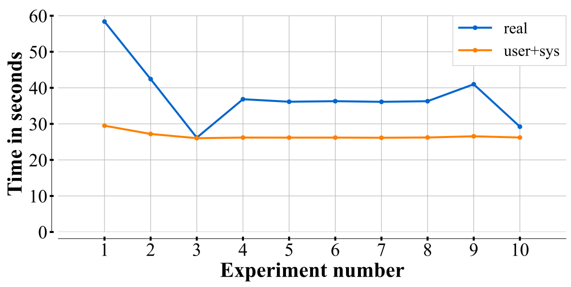
(a) NCP/bilinear; averages: 27.58s / 18.65s



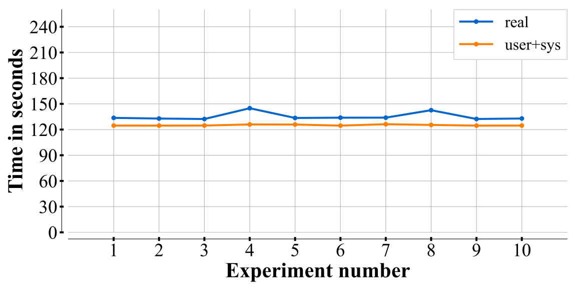
(b) NCP/bicubic; averages: 91.83s / 84.05s

Figure 4.1: Performance of the NCP use cases (30 days) with the updated NESTOR.

has been sped up between 24 and 36 times. While the performance of the bicubic case (Fig. 4.1b) is worse, with an average `real` time of one minute and an half, there is still a significant acceleration: prior to the optimization, such a use case took almost 20 minutes (`user` time), which amounts to more than 13 times slower than the updated NESTOR.

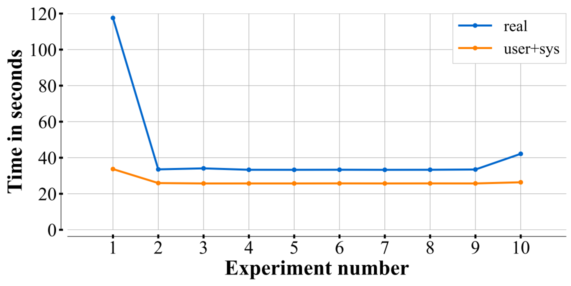


(a) ECP/bilinear; averages: 37.89s / 26.64s

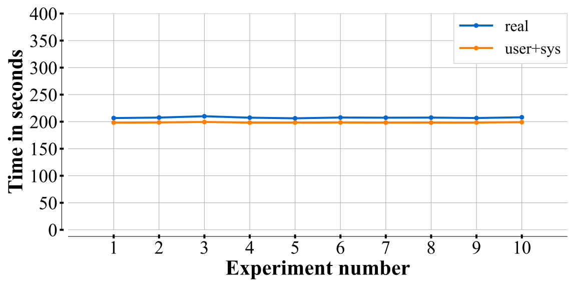


(b) ECP/bicubic; averages: 135.28s / 125.11s

Figure 4.2: Performance of the ECP use cases (30 days) with the updated NESTOR.



(a) ECM/bilinear; averages: 42.74s / 26.62s



(b) ECM/bicubic; averages: 207.57s / 198.63s

Figure 4.3: Performance of the ECM use cases (30 days) with the updated NESTOR.

Figures 4.2 and 4.3 provides figures comparable to those of Figure 4.1 for the ECP and ECM use cases. These use cases are very similar to both NCP use cases: the output domain and time periods are the same and the input domains always consist of regular grids. The main differences are the dimensions of these input domains: for reminders (cf. Table 2.1), ECP features a larger domain than NCP, with 181 by 91 surface cells and 31 vertical levels, while ECM has the largest domain of the initial experiments of Sec. 2.3.1 with 367 by 167 surface cells and 40 vertical levels. Despite these larger dimensions, the bilinear use cases (Fig. 4.2a and Fig. 4.3a) are barely slower than in Fig. 4.1a: around 40 seconds for the `real` times in both cases, and below 30 seconds as for the ideal (`user+sys`) times. These times are also better than initially, especially for the ECP use cases (bilinear is almost 40 times faster), given that the ECM/bilinear case already benefitted from an optimization (cf. Sec. 2.3.2).

The larger dimensions of ECP and ECM grids however appear to have a significant impact on the performance of the bicubic use cases (Fig. 4.2b and Fig. 4.3b). The final times are hopefully still much better than initially, with a bit more than 3 minutes for the ECM/bicubic case versus almost one hour `user` time with the unedited NESTOR (cf. Table 2.1). One possible explanation for this considerable difference is that the search for sampling squares during bicubic interpolation (cf. Sec. 3.2.3) is neither optimized via heuristics [5] nor benefitting from input domain truncation (cf. Sec. 3.2.4): the bigger the input domain, the slower the `bicSet` subroutine.

Listing 4.1: The five most called functions/subroutines in NESTOR with ECP/bicubic (by `gprof`)

```
  %    cumulative    self
time    seconds    seconds    name
42.24      41.78     41.78    spline_
12.87      54.50     12.73    splint_
 5.02      59.46      4.96    __svml_logf4_e9
 4.21      63.62      4.16    bicdo_
 3.85      67.43      3.81    dist_
```

Listing 4.2: The five most called functions/subroutines in NESTOR with ECM/bicubic (by `gprof`)

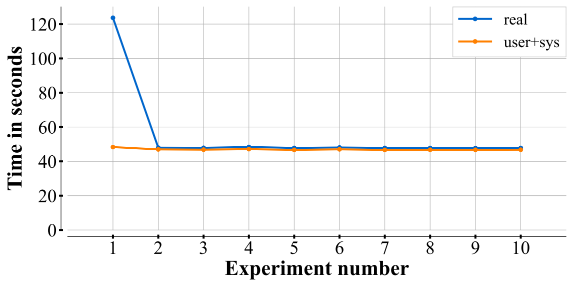
```
  %    cumulative    self
time    seconds    seconds    name
39.56      64.08     64.08    spline_
10.54      81.16     17.08    splint_
 8.83      95.45     14.30    dist_
 6.69     106.29     10.84    __libm_sinf_e7
 6.33     116.54     10.25    __libm_atan2f_e7
```

To assess the hypothesis of `bicSet` slowing down NESTOR, Examples 4.1 and 4.2 provide the five most called functions/subroutines, as determined by the `gprof` tool [4], respectively for the ECP/bicubic and the ECM/bicubic use cases. The times in seconds are not average times but can be considered as representative. In both cases, the most called functions/subroutines are the SPLINE and SPLINT subroutines, which are typically called by `bilDo` (cf. Sec. 3.2.3). This suggests that, regardless of how `bicSet` performs, bicubic interpolation will always be slower than bilinear interpolation because of the need to compute cubic splines all the time. However, both rankings also include the `dist` function, typically involved in the search for sampling squares in `bilSet`. This is especially

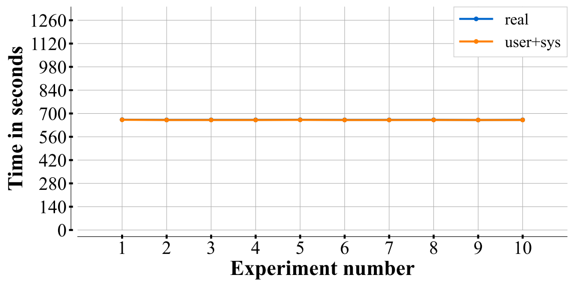
[5] Mostly due to a lack of time; the refreshed bicubic interpolation was amongst the last changes brought to NESTOR.

more visible in Example 4.2, where the fourth and fifth most called functions/subroutines are also trigonometric functions called by `dist`. Moreover, their cumulated times account for more than 30 seconds: this suggests NESTOR may be rendered even faster if a few heuristics (e.g., to take advantage of input domain truncation) were added in `bicSet`.

On a side note, input domain truncation should have a limited effect on ECP and ECM grids: since they cover Europe only, the output domain (a portion of Western Europe notably covering France, Northern Spain, Belgium, etc.) still occupy a significant portion of the input domain. More precisely, the benefits may be less significant than with NCP use cases (dataset covering the entire Earth's surface) or GRD use cases (complete latitude band).



(a) GRD/bilinear; averages: 55.50s / 46.98s



(b) GRD/bicubic; averages: 661.84s / 660.93s

Figure 4.4: Performance of the GRD use cases (31 days) with the updated NESTOR.

Figure 4.4 now depicts the performance of the GRD use cases, with the GRD/bicubic use case having been unassessed with the unedited NESTOR in this work, but being most probably very slow given the dimensions of the input domain. Once again, the performance of the bilinear case (Fig. 4.4a) is clearly below the minute, and even though it constitutes only 4 times less than the time obtained with the unedited NESTOR, it clearly demonstrates the benefits of the changes presented in Chapter 3, and input domain truncation in particular (cf. Sec. 2.3.6). The GRD/bicubic scenario (Fig. 4.4b), on the other hand, is arguably slow, with around 11 minutes spent in all experiments. Like the ECP/ECM use cases, this is likely at least partially due to the lack of heuristics during the search for sampling squares in the `bicSet` subroutine (cf. Sec. 3.2.3), causing NESTOR to explore the entire latitude band.

Listing 4.3: The five most called functions/subroutines in NESTOR with GRD/bicubic (by `gprof`)

```
  %    cumulative    self
 time    seconds    seconds     name
27.74     150.08    150.08     spline_
16.33     238.45     88.37     dist_
12.46     305.85     67.40     __libm_sinf_e7
11.82     369.81     63.96     __libm_atan2f_e7
 9.05     418.77     48.96     __libm_cosf_e7
```

To determine the extent of this last issue, Example 4.3 provides the five most called functions/-subroutines as determined by `gprof` for the GRD/bicubic use case. Here, four of the five most called functions/subroutines are none others than `dist` and the trigonometric functions it calls. This clearly demonstrates a significant amount of time is wasted with `bicSet`, the cumulated times (i.e., those shown in Example 4.3) accounting for around four minutes and an half. Yet, the most called subroutine remains `SPLINE`. This suggests that bicubic interpolation should remain a computationally worse option than bilinear interpolation in all cases, though `NESTOR` (as delivered at the time of writing) may still be able to process such a use case faster as long as a few heuristics are added to `bicSet`. This may cut the total execution time down to a few minutes rather than almost one quarter of an hour. Nevertheless, the times obtained with `GRD`/bicubic remain clearly better than all the times obtained with the unedited `NESTOR` using its own bicubic interpolation (cf. Sec. 2.3.1), which means the updated `NESTOR` outperforms the old version in all cases.

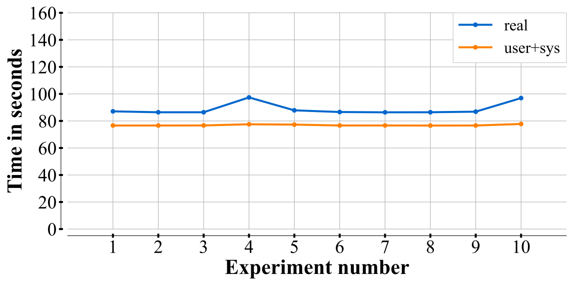


Figure 4.5: Performance of the `MAR` use case (30 days) (averages: 88.84s / 76.90s).

Finally, Figure 4.5 provides the performance for the `MAR` on `MAR` forcing scenario. Due to the interpolation subroutine used for such a case (i.e., `intMAR`, previously named `INTnrg2`, cf. Sec. 3.2.3) being already optimized, the updated `NESTOR` does not outperform the old version as much as with other use cases. Some of the changes proposed in Chapter 3, notably regarding `HWSDsl` and `GLOcov` (cf. Sec. 3.2.2), definitely manage to save several dozens of seconds. However, the most costly operation in this scenario remains the interpolation process itself. Example 4.4 clearly demonstrates that: all five functions/subroutines in the given ranking are involved in the interpolation process tailored for `MAR` on `MAR` forcing. Further optimizing this use case of `NESTOR` would therefore require further optimizing the `intMAR` (formerly `INTnrg2`) subroutine itself.

Listing 4.4: The five most called functions/subroutines in `NESTOR` with `MAR` on `MAR` forcing (by `gprof`)

```
  %    cumulative    self
 time    seconds    seconds     name
16.17      11.12      11.12     dist_
14.39      21.01       9.89     intmar_
11.60      28.98       7.97     __libm_atan2f_e7
11.50      36.88       7.90     __libm_sinf_e7
10.07      43.80       6.92     __libm_cosf_e7
```

4.3 Sensitivity of the MAR model to the updated NESTOR

While the updated NESTOR appears to drastically improve the performance of early 2022's NESTOR while inducing only very small decimal changes in the output files (cf. Sec. 4.2.1), the impact of these small changes on a MAR simulation should be assessed too due to the chaotic nature of meteorological processes [14, 20, 26], as explained in Sec. 4.1. This section therefore evaluates to which extent the MAR results are impacted by the outputs of the new NESTOR by comparing said results to those of a reference simulation (i.e., same settings but with forcing fields computed by the old NESTOR). In other words, this section discusses the ***sensitivity*** of the MAR model to the updated NESTOR.

Sec. 4.3.1 first provides some practical details regarding the settings of the simulations used to evaluate the sensitivity of the MAR model, along with a description of how this sensitivity will be quantified precisely. Sec. 4.3.2 subsequently discusses the sensitivity of the MAR model upon using forcing fields computed by the updated NESTOR with bilinear interpolation. Finally, Sec. 4.3.3 evaluates the effects of using forcing fields prepared with the new bicubic interpolation (cf. Sec. 3.2.3) rather than bilinear interpolation.

4.3.1 Methodology

First of all, to evaluate the sensitivity of the MAR model to the new NESTOR, results from a ***reference simulation*** are required. This simulation must have been run using the output files from January 2022's NESTOR (or an older version) in order to have MAR results that are representative of the previous version(s) of NESTOR. This simulation should be long enough, as decimal changes may not have an impact on the first days of a simulation but rather on the longer term. Several months or a year of simulation should therefore be preferred.

In the context of the CLIM0017 course taught during the first semester of academic year 2021-2022 [10], I had the opportunity to learn how to run the MAR model and managed to run several simulations. The most ambitious of them consisted of simulating the 2001-2010 decade on Scotland and its surroundings. To prepare such a simulation, Fall 2021's NESTOR was run using:

- the NCEP-NCAR Reanalysis 1 dataset [25] for the input domain (144 by 73 cells covering the entire Earth's surface, with 18 levels),
- an output domain of 120 by 120 cells covering 5 by 5 kilometers areas (i.e., 25 square kilometers), centered on the Scottish Highlands,
- and bilinear interpolation.

For illustration purpose, Figure 4.6 pictures the simulated region using the TEX variable found in the output NetCDF file created by NESTOR after tuning it to also create this file. For reminders, this variable corresponds to the soil texture and has 12 possible integer values (0 to 11, included), with 0 corresponding to the sea and sea-level water bodies (colored in purple in Figure 4.6).

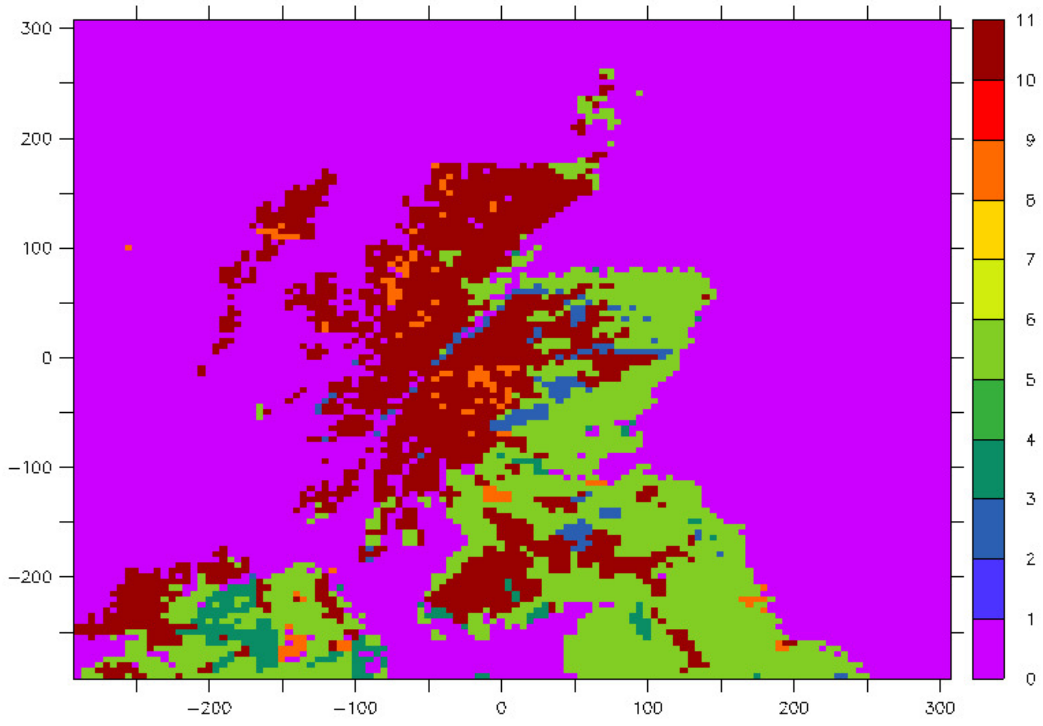


Figure 4.6: Soil texture (`TEX`) for the regional area simulated with the `MAR` model throughout Sec. 4.3. It encompasses mainland Scotland and some of the surrounding isles (including Northern Ireland).

To assess new simulations kickstarted with output files produced by the updated `NESTOR`, the same simulation as above was re-run for the same input/output domains but only for the duration of the year 2001. Indeed, one year is long enough to assess the sensitivity of the `MAR` model, and is also preferable to save time: the aforementioned simulation of the whole 2001-2010 decade indeed took a bit less than four weeks to complete, while simulating one year can be done in 2 or 3 days (depending on the load of the server on which the `MAR` model is running).

Two new simulations with the `MAR` model have been conducted. The first one consisted of simulating 2001 again but using output files produced by the updated `NESTOR` with bilinear interpolation. As such, this simulation is equivalent to the initial one, except the new `NESTOR` is preferred over the old version to prepare the forcing fields. The comparison of both simulations, which will demonstrate whether or not the small decimal changes had a non-negligible impact, is the topic of Sec. 4.3.2.

The second new simulation also consisted of re-simulating 2001 using output files prepared by the updated `NESTOR`, but using bicubic interpolation. Comparing the reference simulation with the last one therefore amounts to evaluating the effects of preparing the `MAR` input data with bicubic interpolation rather than bilinear interpolation, since modern `NESTOR` users typically rely on the former. This second comparison is detailed in Sec. 4.3.3.

Before comparing the results of the three `MAR` simulations, how they can be compared precisely must be established. First of all, six representative variables have been selected to compare the results of the `MAR` model, much like in Sec. 4.2.1.

These six variables are

- TT, the average temperature,
- TTMAX, the maximum temperature,
- TTMIN, the minimum temperature,
- QQ, the specific humidity,
- UV, the wind speed [6],
- and PRC, the sum of MBRR (rainfall) and MBSF (snowfall) [7], which therefore amounts to the total of precipitations.

Each of these variables provides a value for each regional cell and for each day of the year. In fact, the first five variables provide three values for three different levels in sigma coordinates: 0.999 (first level), 0.9994 (second level) and 0.9997 (third level). For convenience, only the level closest to the surface (the third one) will be used to get a single time series of values for each regional cell.

Indeed, by using two sets of time series, respectively for the reference and the new simulation, for each regional cell and for each variable, three common metrics from statistics can be used.

- The **Pearson correlation coefficient** [41] is a normalized correlation coefficient used to evaluate how much two time series correlate, i.e., how similar their evolution are, and typically outputs a value included between -1 and 1. When a coefficient is close to 1, the time series can be considered as evolving in the same manner.
- The **Root-Mean-Square Error** (RMSE) [42] is used to evaluate the magnitude of the differences (or *errors*) between two time series. It is always non-negative and is sensitive to outliers. A small RMSE means the differences (errors) are marginal.
- Finally, the **bias** [32] quantifies the tendency with which a time series deviate from a reference time series. A positive bias, for instance, expresses the fact that the predicted time series tends to have higher values than the reference. Conversely, a negative bias shows that the second time series tends to provide lower values.

The last two metrics can be further divided by the standard deviation of the reference time series to take account of the day-to-day variability. All three metrics can be computed for each regional cell and for each variable, resulting in 2D maps that can show how similar the new simulation is, and if not similar enough, where it differs from the reference and by how much. These maps will be subsequently used in Sec. 4.3.2 and Sec. 4.3.3 to compare the aforementioned MAR simulations.

[6] UVZ, the horizontal wind speed, could also have been used but has been omitted due to space constraints.
[7] PRC does not exist "as is" among the MAR variables. It is computed by the comparison program.

4.3.2 Sensitivity to forcing fields prepared with bilinear interpolation

To compare the results of two `MAR` simulations with the help of the metrics described in Sec. 4.3.1, a small Fortran program has been written to ease the comparison of two NetCDF files with respects to specific variables. For each variable, this program creates a NetCDF file providing three 2D grids with the same dimensions as the regional grid used in the `MAR` simulations, these 2D grids providing respectively, for each regional cell: the correlation coefficient, the root-mean-square error (or RMSE) and the bias (the last two being divided by the standard deviations for each time series in the reference simulation). This program can be found, if necessary, on `climato.be` [8].

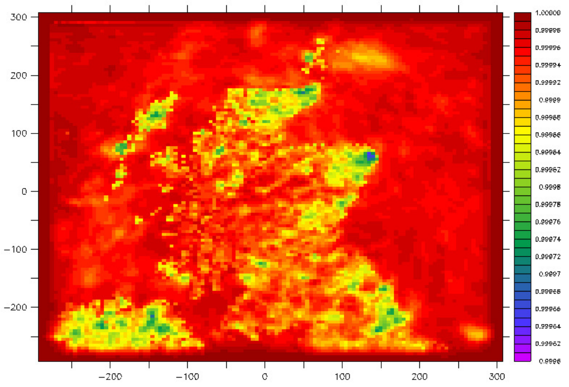
Figure 4.7 first provides the correlation coefficients obtained for all six variables listed in Sec. 4.3.1 between the reference simulation (whose input data has been prepared with the unedited `NESTOR`) and the `MAR` simulation that used output files produced by the updated `NESTOR`. Both versions used bilinear interpolation, which means the settings for both versions of `NESTOR` are identical, though there are small decimal changes in the output files, as previously discussed in Sec. 4.2.1.

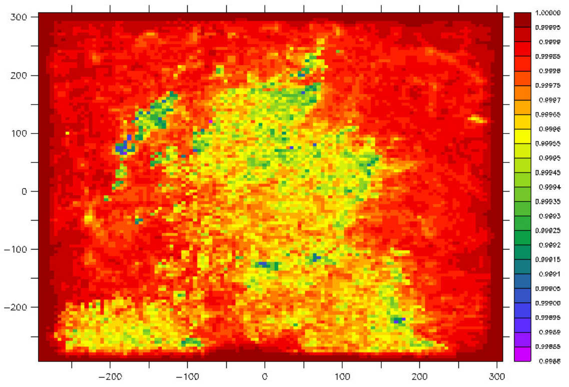
Hopefully, the correlation coefficients are very close to 1 for all variables: each sub-figure of Figure 4.7 is labelled with a lower bound (space constraints prevent having readable scales), with the color scale ranging from this lower bound to 1. Visually speaking, the purple shade is associated with the lower bound while dark red corresponds to 1. With the notable exception of Fig. 4.7f, due to some artifact in the top right corner [9], all coefficients are greater than 0.95, with four of the six figures using a lower bound greater than 0.99. Moreover, the two figures with the lowest bounds, i.e. Fig. 4.7e and Fig. 4.7f, are overwhelmingly (dark) red, meaning the correlation coefficients are very high in most regional cells. All these figures point to the fact that running the `MAR` model with forcing and surface fields as prepared by the updated `NESTOR` had a very negligible impact on the results, as the time series strongly correlate for the vast majority of the regional cells.

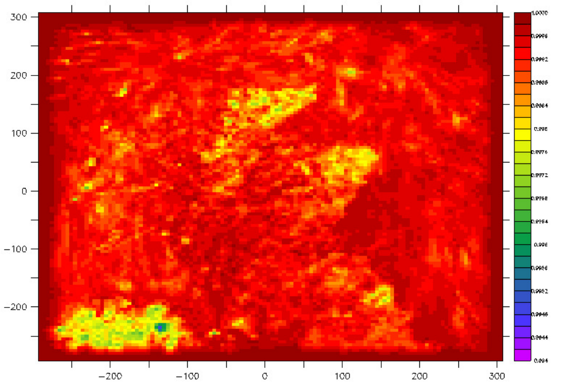
Figure 4.8 now provides the RMSEs for all six variables of interest for the same duo of `MAR` simulations. This time, the sub-figures are labelled with an upper boundary for the RMSE values (sometimes adjusted to have a convenient scale), since the minimum is always 0. For reminders, it is preferrable for the RMSEs to be low, and RMSEs are sensitive to outliers [42]. Again, the results suggest the `MAR` model is barely sensitive to the decimal changes of the updated `NESTOR`: only the wind speed (Fig. 4.8e) and precipitations (Fig. 4.8f) feature seemingly non-negligible RMSEs, i.e., greater than 10^{-1} (though one location in Fig. 4.8c also fits that criterion). These results should however be nuanced. On the one hand, the RMSEs are still low with respects to the units used for both `UV` and `PRC` (m/s and mm, respectively). On the other hand, Fig. 4.8f shows very small RMSEs around the center of the grid (purple areas), which is the area for which the `MAR` model is well tuned when it comes to simulating precipitations. This means that both experiments discussed in this section produced very similar (almost identical) precipitations where they typically should.

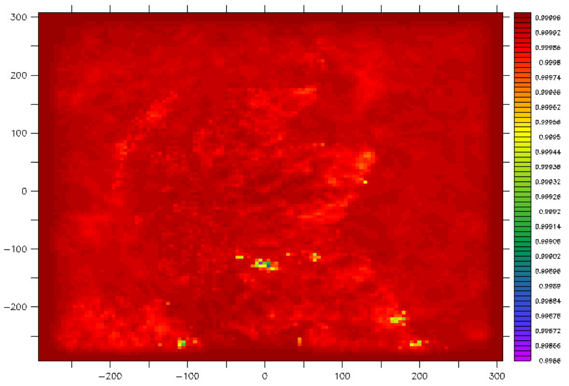
[8] `/srv5_tmp2/jfgrailet/TFE_NESTOR/MAR_output_assessment/original_vs_modified2/comparison.f90`

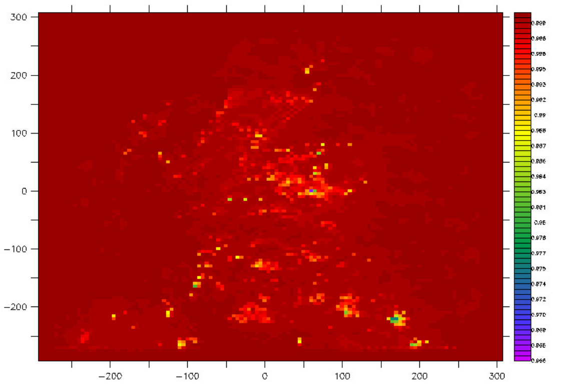
[9] This artifact likely results from how `MAR` proceeds with precipitations. It is typically recommended to expand the borders of a regional grid for the `MAR` to simulate realistic precipitations, because the borders of the regional grid are the locations where clouds start forming in the `MAR` model.


(a) TT (lower bound: 0.9996)


(b) TTMAX (lower bound: 0.9988)


(c) TTMIN (lower bound: 0.994)


(d) QQ (lower bound: 0.9986)


(e) UV (lower bound: 0.966)

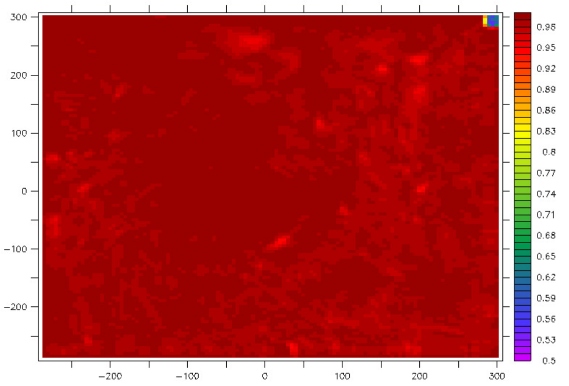
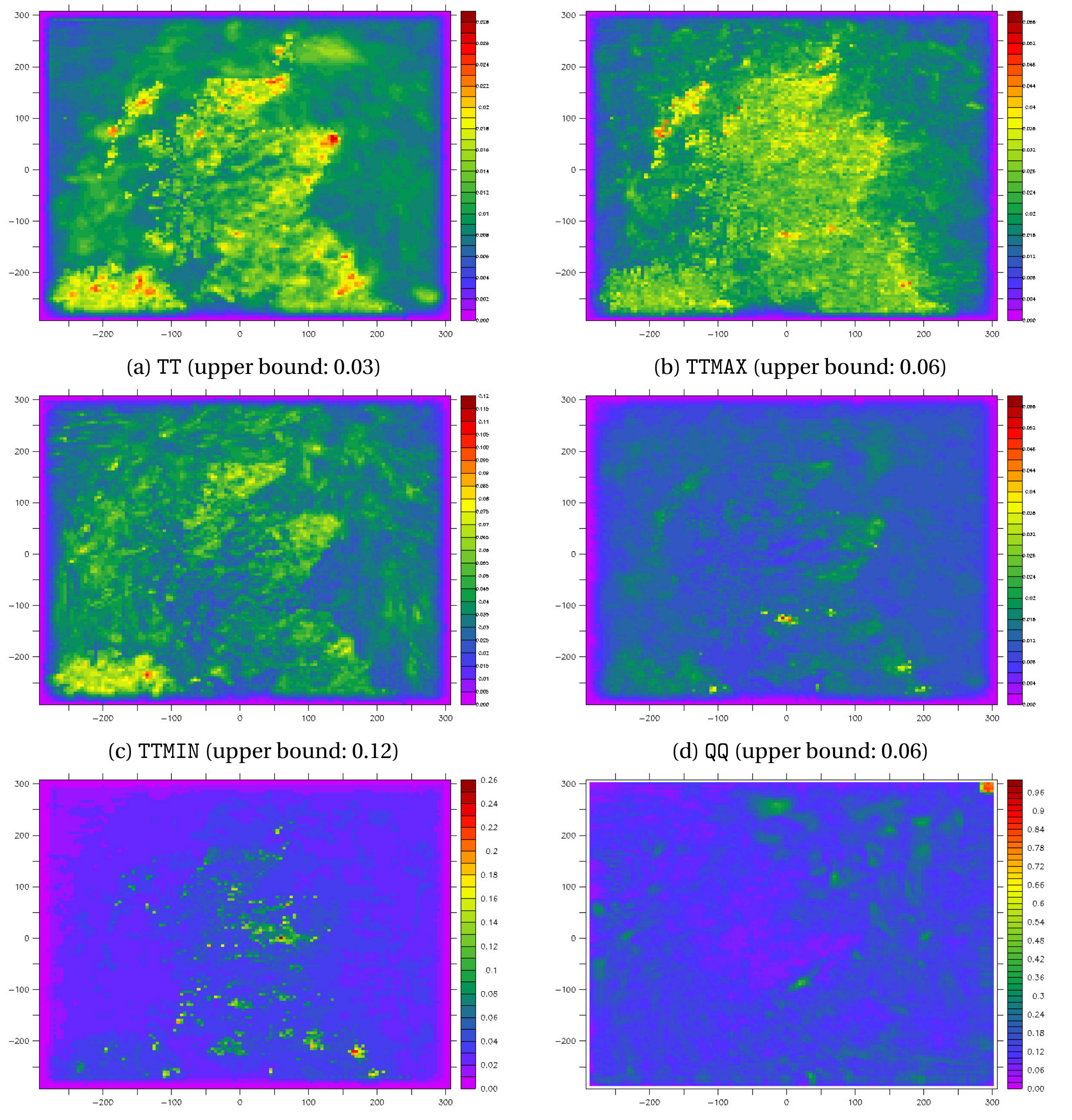

(f) PRC (lower bound: 0.5)

Figure 4.7: Pearson correlation coeffficients for 6 variables between two MAR simulations of the year 2001 over Scotland (prepared with old and new NESTOR, respectively; bilinear interpolation).

Finally, Figure 4.9 provides the biases for the variables of interest in the same manner as Figure 4.7 and 4.8. Each sub-figure is labelled with a symmetric range of values designed to encompass all values contained in each grid. The symmetry of these ranges ensures that the shades of green coincide with bias values close to 0. The very small ranges provided for each figure, combined with the dominant colours of each figure (shades of green), demonstrate that there is little to no bias coming from the MAR simulation prepared with the updated NESTOR with respects to the reference simulation.



(a) TT (upper bound: 0.03)

(b) TTMAX (upper bound: 0.06)

(c) TTMIN (upper bound: 0.12)

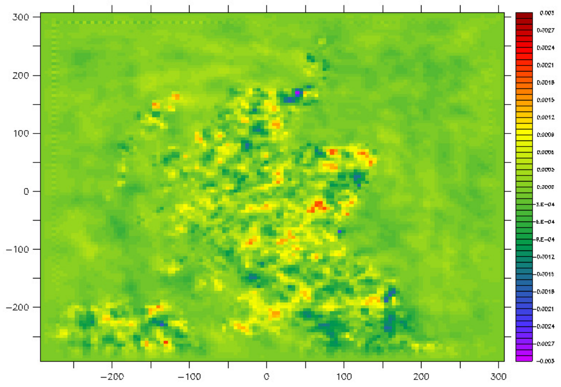
(d) QQ (upper bound: 0.06)

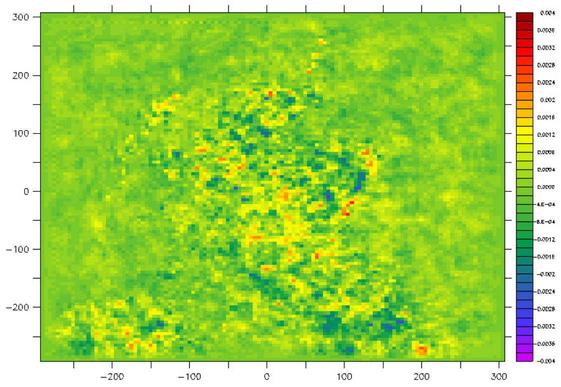
(e) UV (upper bound: 0.26)

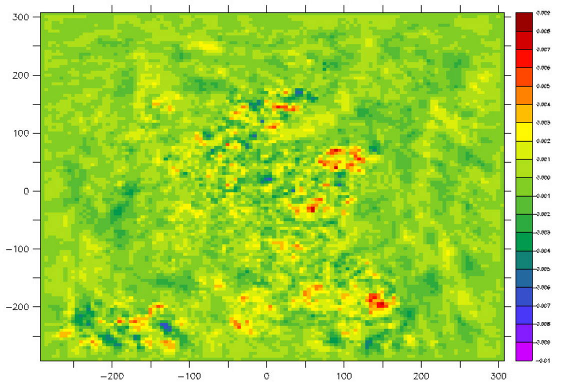
(f) PRC (upper bound: 1)

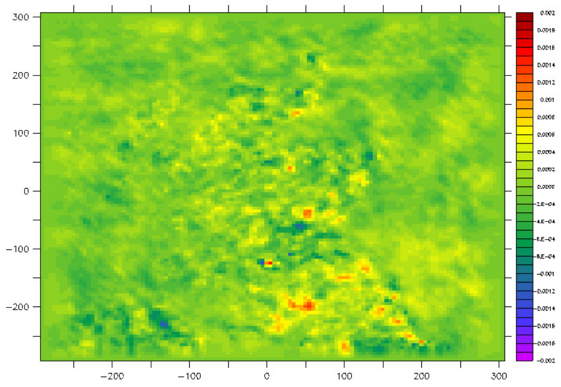
Figure 4.8: RMSEs for 6 variables between two MAR simulations of the year 2001 over Scotland (prepared with old and new NESTOR, respectively; bilinear interpolation).

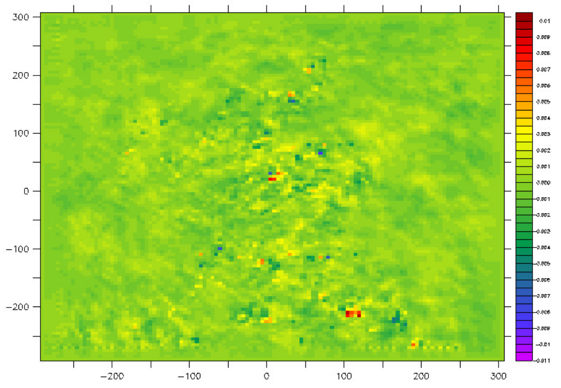
All in all, Figures 4.7, 4.8 and 4.9 demonstrate that using the updated NESTOR did not significantly alter the results of the MAR model. While there are definitely some minor variations, the small errors, near-zero biases and strong correlation coefficients all indicate that the small decimal changes induced by the updated NESTOR, as provided in Table 4.1 in Sec. 4.2.1, had a proportionally negligible impact on the outputs produced by the MAR model. Using the updated NESTOR to prepare new MAR simulations can be therefore considered as safe.


(a) TT (range: -0.003, 0.003)


(b) TTMAX (range: -0.004, 0.004)


(c) TTMIN (range: -0.01, 0.01)


(d) QQ (range: -0.002, 0.002)


(e) UV (range: -0.011, 0.011)

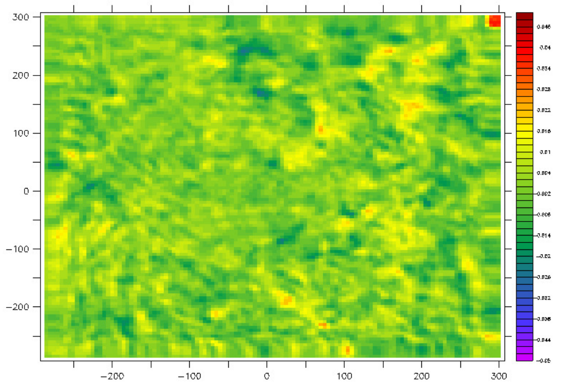
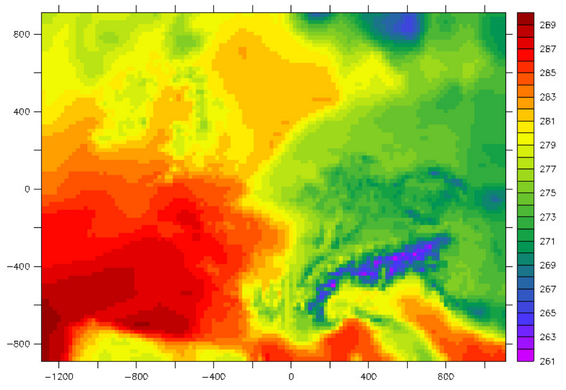

(f) PRC (range: -0.05,0.05)

Figure 4.9: Biases for 6 variables between two MAR simulations of the year 2001 over Scotland (prepared with old and new NESTOR, respectively; bilinear interpolation).

4.3.3 Sensitivity to forcing fields prepared with bicubic interpolation

While the updated NESTOR has been demonstrated to be safe to use despite the small decimal changes induced by the optimized bilinear interpolation (cf. Sec. 4.2.1), nothing suggests the MAR model is not sensitive to forcing and surface fields as prepared with the refreshed bicubic interpolation. After all, bicubic interpolation does not just change a few decimals: as shortly discussed in Chapter 1, the main benefit of bicubic interpolation over bilinear interpolation is that it produces smoother area curves [33, 34]. In other domains of computer science where both types of interpolation are also used to infer high resolution data, such as image processing, bicubic interpolation typically produces finer results. Using NESTOR with bicubic interpolation therefore produces slightly different regional distributions of values (or areas, when visualized) for each variable.


(a) With bilinear interpolation

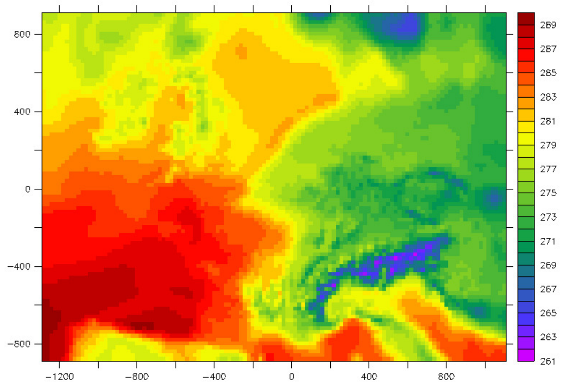
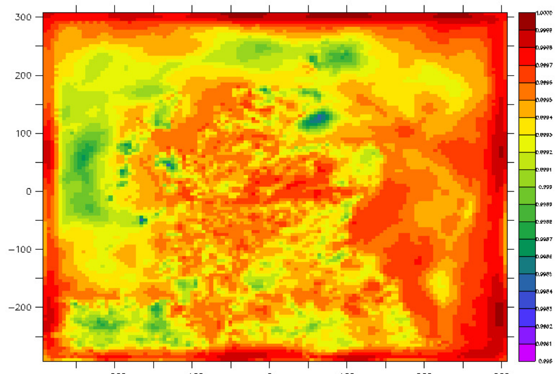

(b) With bicubic interpolation

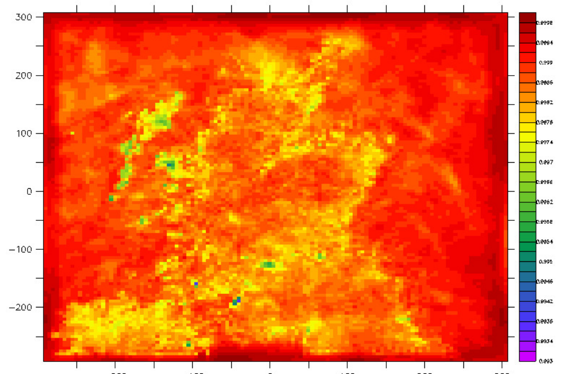
Figure 4.10: Temperature maps (TT; Kelvin) produced by the updated NESTOR with each type of interpolation. ECP data downscaled to Western Europe with 20 by 20 kilometers cells.

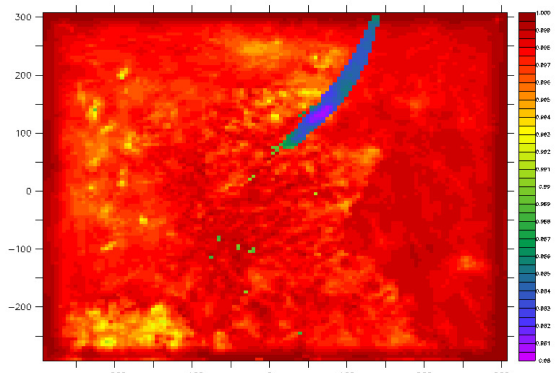
For illustration purpose, Figure 4.10 provides 2D temperature grids (TT variable) for December 4, 2021 at midnight as produced by the updated NESTOR for a regional domain over Western Europe (including France, Belgium, United Kingdom, etc.) with either bilinear (Fig. 4.10a) or bicubic interpolation (Fig. 4.10b). The temperature scale (in Kelvin) is the same in both cases. The input domain was a dataset from the ECMWF (ECP in NESTOR terminology) providing meteorological data over Europe. The bicubic interpolation produces arguably more elegant areas: see for instance how the few isolated pixels in the middle of the top half of Fig. 4.10a appear as better defined areas in Fig. 4.10b. The same goes for various shades of red in the bottom left corner of both maps.

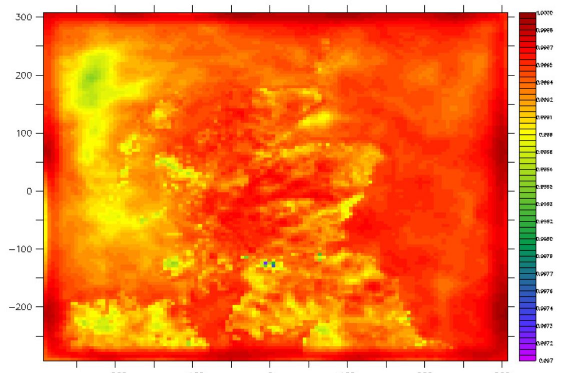
To evaluate the sensitivity of the MAR model to input data prepared with the updated bicubic interpolation (cf. Sec. 3.2.3), a third MAR simulation of 2001 over Scotland has been run using forcing and surface fields produced by the updated NESTOR using bicubic interpolation. The MAR results have then been processed with the same program as described in Sec. 4.3.2, therefore producing three metrics for each regional cell for a total of six variables, as already announced in Sec. 4.3.1.

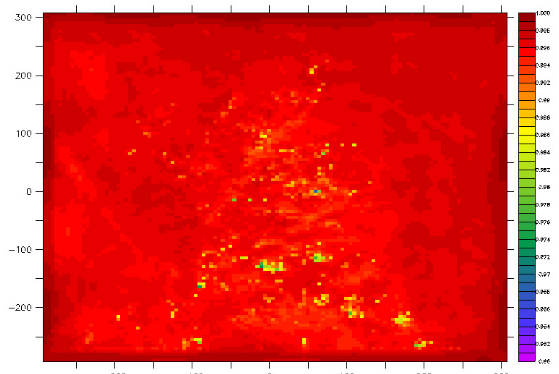
Figure 4.11 first provides the Pearson correlation coefficients obtained for each of the six variables of interest between the reference simulation and the last simulation. The presentation is the same as in Figure 4.7: each sub-figure is labelled with a lower bound corresponding to the lowest possible value represented by the colour scale, the highest value being 1. It is important to remember here that the reference simulation used forcing files prepared by the old NESTOR with bilinear interpolation, knowing bilinear interpolation in the updated NESTOR induces near identical results (cf. Sec. 4.3.3). The main conclusion that can be drawn from Figure 4.11 is that changing the type of interpolation in NESTOR did not significantly change the predictions made by the MAR model: the correlation coefficients remain high for all variables, with five of the six lower bounds being above 0.95 and half of all lower bounds being above 0.99. The main change from Figure 4.7 is the way the correlation coefficients vary: more variations can be observed above the seas surrounding mainland Scotland. Fig. 4.11c (`TTMIN`) also shows a surprising artifact: a tear-like area where the correlation coefficients drop significantly with respects to the rest of the map, though they remain high (higher than 0.98). Interestingly, this artifact only appears for the level closest to the surface of `TTMIN`: recomputing the maps with either of the other two levels does not produce such an artifact.


(a) `TT` (lower bound: 0.998)


(b) `TTMAX` (lower bound: 0.993)


(c) `TTMIN` (lower bound: 0.98)


(d) `QQ` (lower bound: 0.997)


(e) `UV` (lower bound: 0.96)

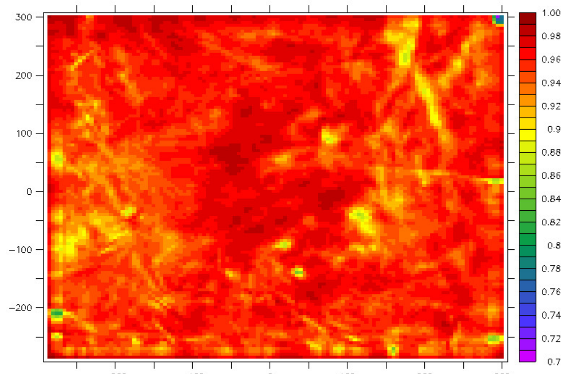
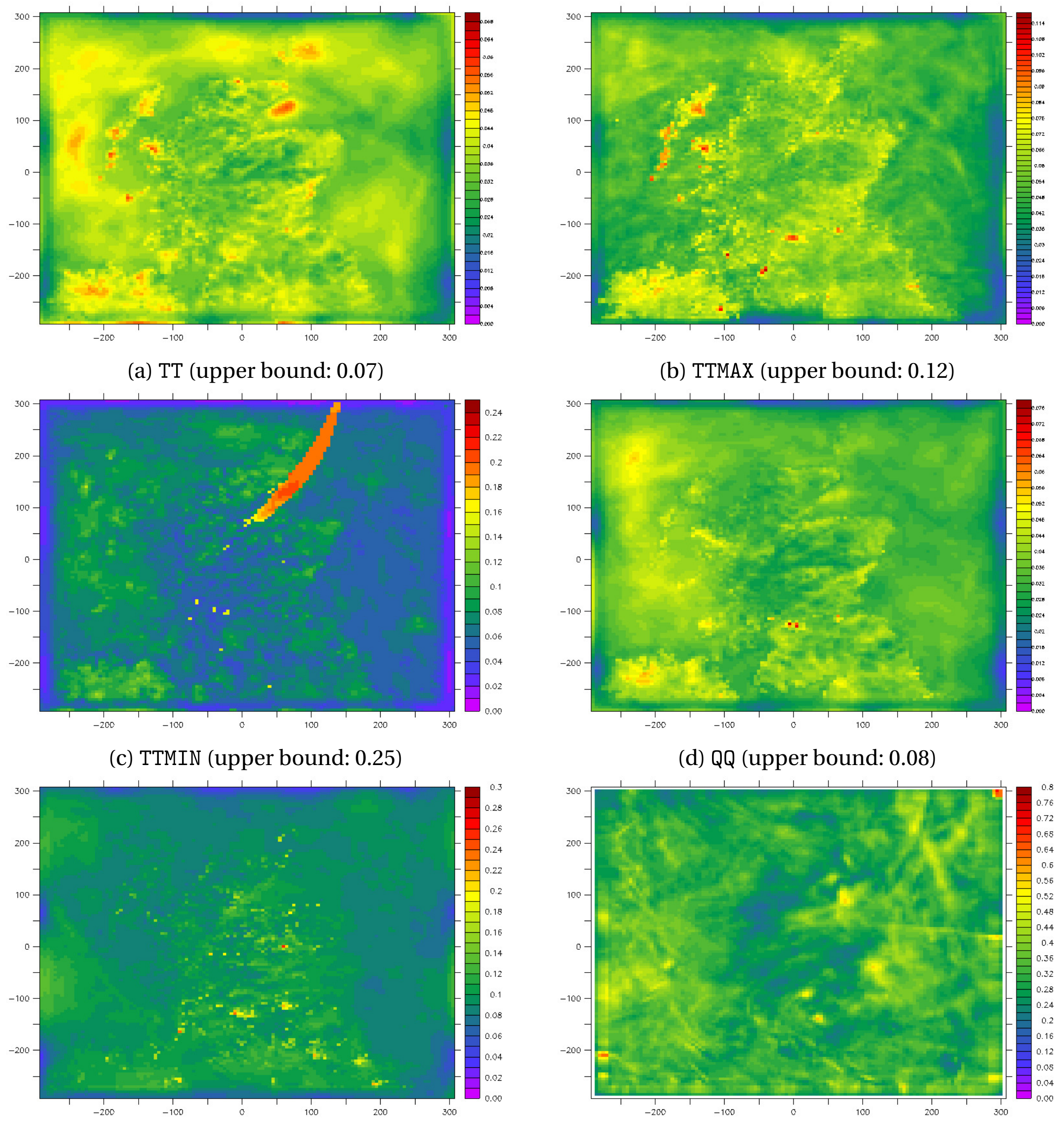

(f) `PRC` (lower bound: 0.7)

Figure 4.11: Pearson correlation coeffficients for 6 variables between two `MAR` simulations of the year 2001 over Scotland (with old and new `NESTOR`, respectively using bilinear and bicubic interpolation).

To complement the previous figure, Figure 4.12 provides the root-mean-square errors (RMSEs) for the same duo of simulations with the same presentation as before. Like Figure 4.8, each sub-figure is labelled with an upper bound on the RMSE. Again, these new figures suggest the `MAR` model did not behave much differently than in Sec. 4.3.2: the RMSEs are slightly higher, but the upper bounds used to represent all of them have been a bit more than doubled at worst.



(a) `TT` (upper bound: 0.07)

(b) `TTMAX` (upper bound: 0.12)

(c) `TTMIN` (upper bound: 0.25)

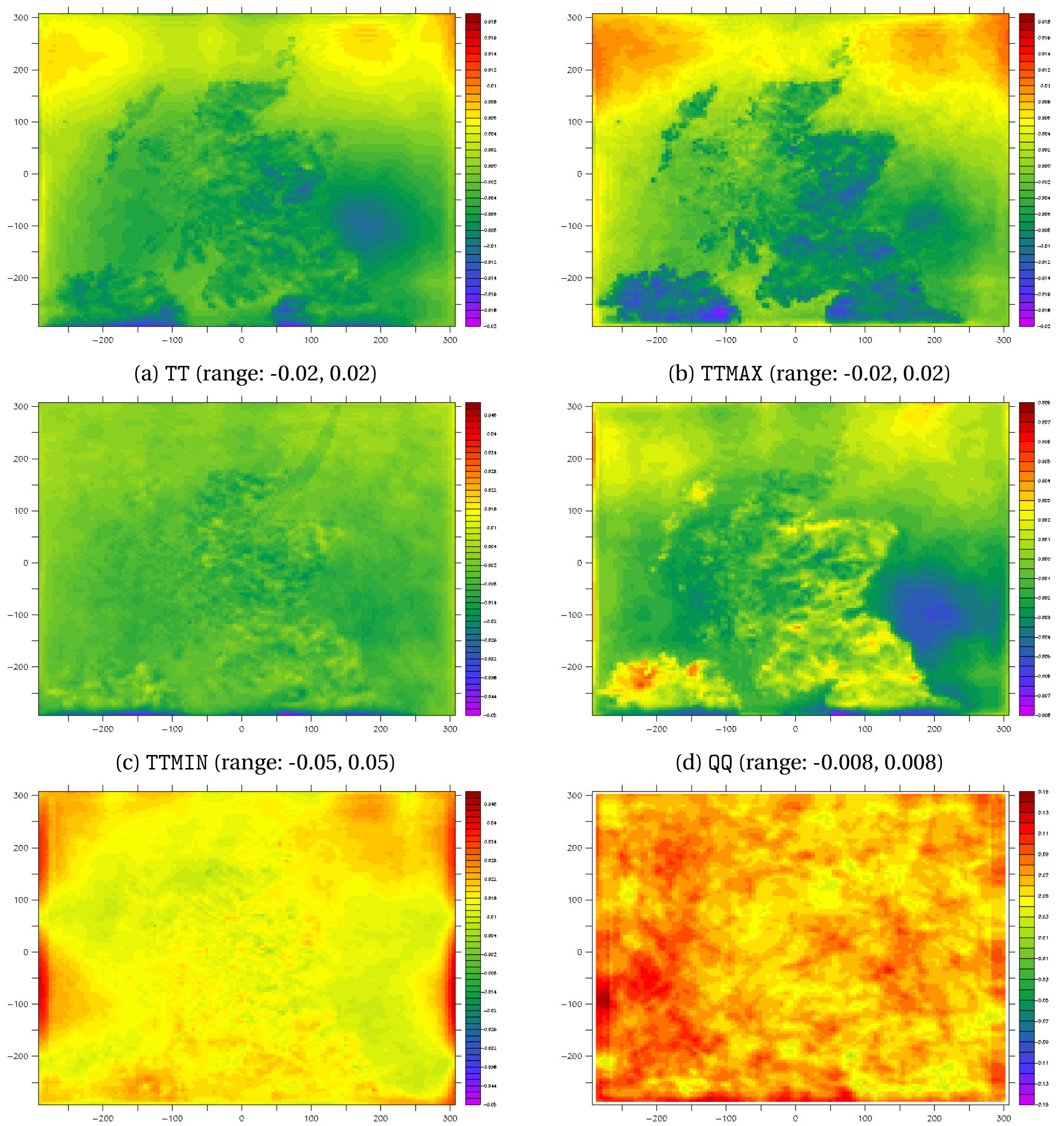
(d) `QQ` (upper bound: 0.08)

(e) `UV` (upper bound: 0.3)

(f) `PRC` (upper bound: 0.8)

Figure 4.12: RMSEs for 6 variables between two `MAR` simulations of the year 2001 over Scotland (with old and new `NESTOR`, respectively using bilinear and bicubic interpolation).

Like in Figure 4.11, there are also more variations above the seas in Figure 4.12, and a tear-like artifact also appears for `TTMIN` in Fig. 4.12c. This last observation is however not surprising, given that this visual artifact corresponds to a sudden drop in the correlation coefficients in Fig. 4.11c: due to the RMSE being sensitive to outliers [42], it was bound to appear too.



(a) `TT` (range: -0.02, 0.02)

(b) `TTMAX` (range: -0.02, 0.02)

(c) `TTMIN` (range: -0.05, 0.05)

(d) `QQ` (range: -0.008, 0.008)

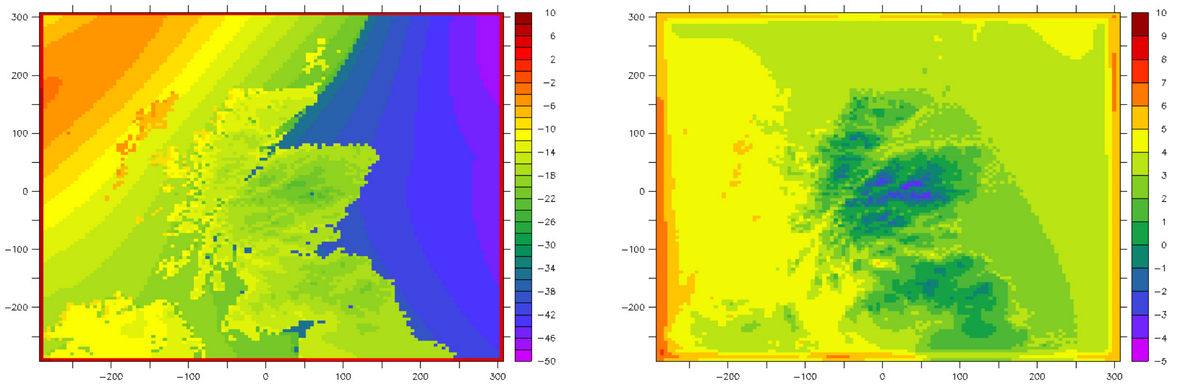
(e) `UV` (range: -0.05, 0.05)

(f) `PRC` (range: -0.15,0.15)

Figure 4.13: Biases for 6 variables between two `MAR` simulations of the year 2001 over Scotland (with old and new `NESTOR`, respectively using bilinear and bicubic interpolation).

Finally, Figure 4.13 provides the biases obtained with the same duo of simulations, presented in the same way as Figure 4.9, i.e., with symmetric ranges of values for each sub-figure. This last figure highlights the differences that are induced by the bicubic interpolation in a much clearer way than previously. While Figure 4.9 in Sec. 4.3.2 suggested there was little to no bias between two MAR simulations kickstarted respectively with the old and the updated NESTOR (both using bilinear interpolation), Figure 4.13 shows slight biases above the seas surrounding Scotland. In particular, the MAR model seems to have predicted slightly higher temperature values in the north of the maps for TT (Fig. 4.13a) and TTMAX (Fig. 4.13b), and conversely, colder temperatures on the eastern coast of Scotland. The QQ map (Fig. 4.13d) follows the same trend. Finally, the maps obtained for UV (Fig. 4.13e) and PRC (Fig. 4.13f) show a tendency to predict slightly stronger winds and a bit more precipitations. Interestingly, the TTMIN map (Fig. 4.13c) suggest there is little to no bias for this variable.

The biases for surface levels highlighted by Figure 4.13 tends to suggest the MAR model is slightly sensitive to whether NESTOR uses bilinear or bicubic interpolation. The latter seems to have had an impact on marine areas in the context of the regional setting used throughout this section (Scotland and the surrounding isles). However, these observations must be nuanced in at least two ways. On the one hand, the observed biases and RMSEs remain low, meaning we are dealing here with only subtle variations with respects to the reference simulation. On the other hand, the variations over marine areas may depend on the regional setting: to have a better picture of the effects of the new bicubic interpolation, additionnal MAR simulations should be run over other regional settings featuring either more, either less land areas. Simulating regional settings with a soft topography (including seas or not) should also be considered: Scotland being a mountainous area, its topography may have played a role in the distribution of the biases in Figure 4.13.



(a) σ = 0.9997 (range: -50, 10) (b) σ = 0.9994 (range: -5, 10)

Figure 4.14: Bizarre TTMIN maps for the first day of the third MAR simulation of 2001 over Scotland.

Finally, the tear-like artifact shown in Fig. 4.11c and Fig. 4.12c (that is also barely visible in Fig. 4.13c) deserves some comments. For reminders, this artifact only occurred at the level of TTMIN that was the closest to the surface (σ = 0.9997), other levels (σ = 0.9994 and σ = 0.999) showing no particular anomaly. After a quick look into the output files produced by the third MAR simulation, it turned out that the very first day of 2001 for the TTMIN variable [10] provided a very unusual map

[10]The MAR NetCDF files, in this context, provide grids for each day and each variable.

for the surface level: there was indeed a very sharp decrease in TTMIN towards the east, partially coinciding with the tear-like artifact, with a thermal amplitude of around 60 degrees. Looking at the level above, surprisingly, provided a perfectly sound map whose thermal amplitude was a bit less than 15 degrees. At the time of writing, it is unclear why the MAR model produced such a result for the first day, especially since there seemed to be no particular issue with the files provided by the updated NESTOR, at least regarding the TT and PT variables for the first day of 2001. Whether it is a NESTOR or a MAR issue and how it can be fixed is left for future work. The two aforementioned levels of TTMIN for the first day of 2001 are pictured in Figure 4.14 for illustration purpose.

4.4 Closing comments: small changes for the code, a big step for NESTOR

The multiple evaluations performed throughout this chapter have led to two major results. On the one hand, the updated NESTOR is significantly more performant than the old version, the former being up to 40 times faster than the latter, notably with a use case involving bilinear interpolation and the ECP data source (cf. Sec. 4.2.2). Only the case of the MAR on MAR forcing appears to have been only marginally improved, as the constraining factor of this use case remains the interpolation process, which has been already optimized (cf. Example 4.4 in Sec. 4.2.2). Otherwise, the various changes introduced in Chapter 3 all allow the updated NESTOR to divide the execution times of the old version several times. For instance, the execution time of the Gulf of Alaska scenario (introduced in Sec. 2.3.6) could be divided by four notably thanks to the idea of truncating the input domain before preparing its vertical levels (cf. Sec. 4.2.2), this use case being already arguably fast by the standards of the old NESTOR thanks to some optimization for the ECM data source (cf. Sec. 2.3.3).

On the other hand, a new MAR simulation demonstrated that small decimal changes induced by the optimizations of the interpolation library (cf. Sec. 3.2.3 and Sec. 4.2.1) did not result in significant differences, the new simulation showing near-perfect correlation with a reference simulation, along with small root-mean-square errors [42] and little to no bias. This shows that using the updated NESTOR to prepare new MAR simulations is safe, at the very least with bilinear interpolation.

Indeed, the new bicubic interpolation (cf. Sec. 3.2.3), while being significantly faster and more realistically usable than before (cf. Sec. 4.2.2), may require some more work in order to assess thoroughly its impact on the MAR model. A third MAR simulation involving this new bicubic interpolation has demonstrated that there are some positive and negative biases within sub-areas of the regional setting. However, these biases may change in other settings. It also seemed this last simulation also predicted slighty stronger winds and more precipitations, though the changes remain subtle. Regardless of its impact on the MAR model, the new bicubic interpolation itself may also need some more work to optimize the search for sampling squares and therefore speed up typical use cases involving it, though it is clear, at this point, that running NESTOR with bicubic interpolation will always be slower than with bilinear interpolation (cf. Sec. 4.2.2).

Chapter 5

Conclusion

Simulating the Earth's climate is a challenge that is as crucial as it is complex. Not only computer climate models must accurately simulate the physical processes occurring in the atmosphere and at the interface between this same atmosphere and the surface [13, 14, 20, 23], but they should also be computationally efficient to obtain predictions regarding the future of our climate in a reasonable time (or in an exhaustive manner), both at the global and the regional scales. The MAR model [21], for ***M**odèle **A**tmosphérique **R**égional*, is a good example of regional computer climate model that is particularly effective at handling precipitations, snow and ice and which is actively used by several research groups to study polar regions and hydroclimatic regimes [1, 12, 16, 43].

To be able to simulate regional climates in a realistic manner, the MAR model requires ***forcing fields***, i.e., climatic data that is fed to the borders of the regional domain at specific time steps in order to take account of the evolution of the climate outside the region of interest. This data comes either from real-life measurements or from a simulation run with a global climate model. Regardless of where the data comes from, it first has to be processed by a companion software of the MAR model called NESTOR [22] (***NEST**ing **OR**ganization for the preparation of meteorological and surface fields*). NESTOR receives NetCDF files [39] providing various grids (for temperature, specific/relative humidity, etc.) associated to a large scale domain (e.g., covering the entire Earth's surface or a continent, like the ERA-5 dataset [9]) and downscales said grids into regional grids that the MAR model will use both for its initialization and for boundary forcing throughout the simulation. In practice, this consists of inferring high resolution data from low resolution grids through various interpolation processes, most notably bilinear and bicubic interpolation [33, 34] [1].

Despite being essential to start simulations with the MAR model, NESTOR has not been maintained as thoroughly as the MAR model itself. Its main source file NESTOR.f still advertises the user that the

[1]Some subroutines of NESTOR do not rely on neither bilinear nor bicubic interpolation. See for instance HWSDsl.

current version was released around June 2004, though a few components of early 2022's NESTOR [22] have been added much more recently, such as the HWSDsl and GLOcov subroutines (used to downscale soil texture and soil cover data). While NESTOR still does the job it is intended to do, its lack of maintenance over time and its poor performance constitute additional constraints to MAR users. For instance, in some typical use cases, NESTOR may require between 15 to 30 minutes (sometimes more) to process one month of MAR input data. As a result, a scientist planning a MAR simulation over several months or years may have to delay the start of this simulation or start it even before NESTOR finished preparing the boundary forcing fields. Both situations are problematic for performance and call for an update of NESTOR at the very least improving its performance.

The purpose of this master thesis was to review the source code of early 2022's NESTOR to pinpoint where it wasted time and to design a solution for each performance issue. The main outcome of this work is **an updated NESTOR that divides several times the execution times of the previous version**, with numerous typical use cases now completing in (much) less than a minute (cf. Sec. 4.2.2). The greatest acceleration observed is 40 times faster than with the initial version (i.e., as of January 2022). This significant outcome results from the three following contributions.

- First of all, the **main computational flaws of NESTOR have been characterized and quantified** to determine where NESTOR was losing time the most. This first contribution is important because future programmers wishing to further update NESTOR may need to know what were its typical flaws prior to this work. For instance, a programmer tasked with fully refactoring NESTOR may want to learn more about I/O calls (i.e., in this context, reading NetCDF or control files), as it was demonstrated a lot of time was wasted by early 2022's NESTOR by making an unnecessary large number of such calls. This detailed analysis can be found in Chapter 2.

- Second, **practical solutions to tackle these flaws have been designed and implemented**. Each of these solutions was pragmatic and did not require to significantly edit the source code. At worst, some groups of files (subroutines) have been re-organized, but the updated NESTOR still features the same program architecture and coding style. On top of that, the updated NESTOR also cleans the former version from obsolete features and slightly improves code readability. The details regarding these changes can be found in Chapter 3.

- Third, new MAR simulations demonstrated that **the updated NESTOR leads to MAR results that are consistent with those of past simulations started with the old NESTOR**. Indeed, the updated NESTOR produce forcing files whose real variables exhibit small decimal changes with respects to what the old NESTOR produced (cf. Sec. 4.2.1). In climate science, this may be an issue because of the well-known chaotic behaviour of some meteorological processes [20, 23, 26]. Hopefully, practical experiments with both the old and new NESTOR and the MAR model demonstrated these decimal changes have a negligible impact on the results. This holds also true upon using the refreshed bicubic interpolation, though it still induces some variations. This last contribution is detailed in Chapter 4.

Moreover, two topics discussed in this work may be further deepened in the future.

- As highlighted many times through Chapter 4, the idea of input domain truncation may be generalized to other algorithmic steps of `NESTOR`. In the updated version, this idea is only applied to the preparation of vertical levels because this was one of the steps where `NESTOR` wasted the most time upon downscaling a large input domain (such as the one discussed in Sec. 2.3.6), but the same idea could also be applied to other areas. For example, the memory allocations used in the refreshed `HWSDsl` and `GLOcov` subroutines could be reduced with this idea, and the search for sampling squares during the very first bicubic interpolation (new version) could be optimized by restricting said search to the truncated domain.
- Speaking of bicubic interpolation, the sensitivity of the `MAR` model to forcing files produced with bicubic interpolation rather than bilinear interpolation may be studied in greater details (e.g., with other regional settings) to assess whether or not bicubic interpolation has benefits. Indeed, a first evaluation provided in Sec. 4.3.3 showed that bicubic interpolation induces some small biases in the `MAR` output files with respects to output files coming from a `MAR` simulation initially prepared with bilinear interpolation. Since running `NESTOR` with bicubic interpolation should always be slower than upon using bilinear interpolation, potential benefits of the former should be characterized to give an incentive for keeping this feature.

Finally, there are a few other ways to improve `NESTOR` that have not been investigated yet due to a lack of time. These ways may be explored by future programmers to either ease the maintenance of `NESTOR` or to improve its main features.

- For readability's sake, the source code of `NESTOR` should be re-formatted in "*free form*" Fortran, the current code still being formatted according to the old "*fixed form*" conventions [37]. Other old conventions (e.g., using `.eq.` to test equality rather than the more common ==) may be updated to improve the readability of the source code to future programmers.
- While this work did some cleaning regarding obsolete features (cf. Sec. 2.4.2 and Chapter 3), there may still be a few subroutines or sets of instructions that could be simplified or removed altogether. To achieve this, future programmers could make an exhaustive inventory of the features offered by `NESTOR` and which one are still benefitial to modern users.
- As neither the old nor the updated `NESTOR` takes advantage of parallel computing (i.e., using multiple CPUs), it could be interesting to re-design the main loop of `NESTOR` (cf. Sec. 2.1) such that each available CPU processes a different iteration. This is feasible notably because producing regional grids at a specific iteration does not depend on values computed at the previous iteration(s). One way to achieve this would consist of performing the initialization steps of `NESTOR` on a single CPU, along with the very first iteration (in order to load the sampling data used to optimize subsequent interpolations, cf. Sec. 3.2.3), then share the subsequent

iterations between multiple CPUs. Another way to take advantage of parallel computing would consist of parallelizing operations within a single iteration of the main loop (i.e., in the `NSTint` subroutine). No matter the selected approach, future updates of `NESTOR` could definitely benefit from parallel computing to improve its performance.

- Finally, while this work fixed in `NESTOR` the option for using cubic splines [35] to interpolate vertical levels (the same option in the old `NESTOR` produced broken NetCDF files, cf. Sec. 2.4.2), the benefits of this approach with respects to the traditional linear interpolation (as performed by `INTlin` in the interpolation library `INTERp.f`) have not been assessed. In practice, this is only due to a lack of time. Some early practical tests suggest, however, that there is little to no difference between both types of interpolation.

In conclusion, while there are still many ways to improve `NESTOR`, the updated version presented in this work should definitely save time to `MAR` users thanks to the significantly improved performance. While performance may still be improved for specific use cases (in particular those involving bicubic interpolation), future updates of `NESTOR` may focus on code readability and maintenance, at least to ensure `NESTOR` is eventually as well maintained as the `MAR` model itself [21].

BIBLIOGRAPHY

[1] C. AMORY, C. KITTEL, L. LE TOUMELIN, C. AGOSTA, A. DELHASSE, V. FAVIER, AND X. FETTWEIS, *Performance of MAR (v3.11) in simulating the drifting-snow climate and surface mass balance of Adélie Land, East Antarctica*, Geoscientific Model Development, vol. 14, pp. 3487–3510, June 2021.

[2] V. BREÑA-MEDINA, *University of Bristol Thesis Template*.
`https://fr.overleaf.com/latex/templates/university-of-bristol-thesis-template/kzqrfvyxxcdm/`.
Accessed: May 25, 2022.

[3] CLIMATE PREDICTION CENTER (NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION), *GrADS tutorials*.
`https://www.cpc.ncep.noaa.gov/products/international/usrcc/grads.shtml`.
Accessed: June 3, 2022.

[4] COMMANDLINUX.COM, *GPROF*.
`https://www.commandlinux.com/man-page/man1/gprof.1.html`.
Accessed: June 3, 2022.

[5] ——, *TIME*.
`https://www.commandlinux.com/man-page/man1/time.1.html`.
Accessed: June 3, 2022.

[6] CREATIVE COMMONS, *Attribution 4.0 International (CC BY 4.0)*.
`https://creativecommons.org/licenses/by/4.0/deed.en`.
Accessed: May 25, 2022.

[7] K. DE RIDDER AND G. SCHAYES, *The IAGL land surface model*, Journal of Applied Meteorology - J APPL METEOROL, vol. 36, pp. 167–182, February 1997.

[8] E. ELTAHIR AND A. KROL, *MIT Climate Portal - Climate Models*.
`https://climate.mit.edu/explainers/climate-models`, January 2021.
Accessed: May 30, 2022.

[9] EUROPEAN CENTRE FOR MEDIUM-RANGE WEATHER FORECASTS, *Era5 | ecmwf.* `https://www.ecmwf.int/en/forecasts/datasets/reanalysis-datasets/era5`. Accessed: June 1, 2022.

[10] X. FETTWEIS, *CLIM0017 - Modélisation du climat*, September 2021.

[11] ——, *GEOG2020 - Éléments de météorologie*, March 2022.

[12] X. FETTWEIS, B. FRANCO, M. TEDESCO, J. VAN ANGELEN, J. LENAERTS, M. VAN DEN BROEKE, AND H. GALLÉE, *Estimating the Greenland Ice Sheet surface mass balance contribution to future sea level rise using the regional atmospheric climate model MAR*, The Cryosphere, vol. 7, pp. 469–489, March 2013.

[13] L. FRANÇOIS, *CLIM0002 - Climate Modelling*, October 2021.

[14] ——, *SPAT0024 - Météorologie*, September 2021.

[15] D. GOLDBERG, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, Computer Surveys, March 1991.
Available in its entirety at: `https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`.

[16] E. HANNA, J. CAPPELEN, X. FETTWEIS, S. H. MERNILD, T. L. MOTE, R. MOTTRAM, K. STEFFEN, T. J. BALLINGER, AND R. J. HALL, *Greenland surface air temperature changes from 1981 to 2019 and implications for ice-sheet melt and mass-balance change*, International Journal of Climatology, vol. 41, no. S1, pp. E1336–E1352, July 2020.

[17] INTEL COMMUNITIES, *Inefficient code for simple string comparison.* `https://community.intel.com/t5/Intel-Fortran-Compiler/Inefficient-code-for-simple-string-comparison/m-p/936478/`. Forum post accessed on June 7, 2022.

[18] L. LEUNG, *Regional Climate Models*, January 2012.

[19] R. M. MACKAY AND M. A. K. KHALIL, *Theory and development of a one dimensional time dependent radiative convective climate model*, Chemosphere, vol. 22, no. 3, pp. 383–417, 1991.

[20] S. MALARDEL, *Fondamentaux de Météorologie, 2e édition*, Éditions Cépaduès, 2009.

[21] MAR GROUP, *MAR GitLab Repository.* `https://gitlab.com/Mar-Group/MARv3/`. Requires a GitLab account.

[22] ——, *NESTOR GitLab Repository.*
https://gitlab.com/Mar-Group/MARv3/-/tree/master/src/NESTOR/.
Requires a GitLab account.

[23] K. McGuffie and A. Henderson-Sellers, *The Climate Modelling Primer*, Wiley, 2014.

[24] NOAA, *Atmospheric Model (Schematic).*
http://celebrating200years.noaa.gov/breakthroughs/climate_model/AtmosphericModelSchematic.png, November 2007.
Found on: https://commons.wikimedia.org/wiki/File:AtmosphericModelSchematic.png.

[25] NOAA Physical Sciences Laboratory, *NCEP/NCAR Reanalysis 1: Summary.*
https://psl.noaa.gov/data/gridded/data.ncep.reanalysis.html.
Accessed: May 31, 2022.

[26] E. Norton Lorenz, *Deterministic nonperiodic flow*, Journal of the Atmospheric Sciences, vol. 20, pp. 130–141, 1963.

[27] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in FORTRAN 77: The Art of Scientific Computing*, Cambridge University Press, September 1992.

[28] A. Silberschatz, G. Gagne, and P. Galvin, *Operating System Concepts (8th edition)*, Wiley, 2011.
See Chapter 13 and especially figure 13.10.

[29] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts (10th edition)*, Wiley, 2018.
See Chapter 11.

[30] The Craft of Coding, *Floating-point precision in Fortran.*
https://craftofcoding.wordpress.com/2020/01/27/floating-point-precision-in-fortran/, January 2020.
Accessed: June 12, 2022.

[31] The Fortran Wiki, *Real precision.*
https://fortranwiki.org/fortran/show/Real+precision.
Accessed: June 12, 2022.

[32] Wikipedia, *Bias of an estimator.*
https://en.wikipedia.org/wiki/Bias_of_an_estimator.
Accessed: June 13, 2022.

[33] ——, *Bicubic interpolation.*
`https://en.wikipedia.org/wiki/Bicubic_interpolation`.
Accessed: May 30, 2022.

[34] ——, *Bilinear interpolation.*
`https://en.wikipedia.org/wiki/Bilinear_interpolation`.
Accessed: May 30, 2022.

[35] ——, *Cubic Hermite spline.*
`https://en.wikipedia.org/wiki/Cubic_Hermite_spline`.
Accessed: May 30, 2022.

[36] ——, *Earth.*
`https://en.wikipedia.org/wiki/Earth`.
Accessed: June 1, 2022.

[37] ——, *Fortran - Fixed layout and punched cards.*
`https://en.wikipedia.org/wiki/Fortran#Fixed_layout_and_punched_cards`.
Accessed: June 1, 2022.

[38] ——, *Linear interpolation.*
`https://en.wikipedia.org/wiki/Linear_interpolation`.
Accessed: May 30, 2022.

[39] ——, *NetCDF.*
`https://en.wikipedia.org/wiki/NetCDF`.
Accessed: May 30, 2022.

[40] ——, *North atlantic oscillation.*
`https://en.wikipedia.org/wiki/North_Atlantic_oscillation`.
Accessed: June 1, 2022.

[41] ——, *Pearson correlation coefficient.*
`https://en.wikipedia.org/wiki/Pearson_correlation_coefficient`.
Accessed: June 13, 2022.

[42] ——, *Root-mean-square deviation.*
`https://en.wikipedia.org/wiki/Root-mean-square_deviation`.
Accessed: June 13, 2022.

[43] C. Wyard, C. Scholzen, S. Doutreloup, E. Hallot, and X. Fettweis, *Future evolution of the hydroclimatic conditions favouring floods in the south-east of Belgium by 2100 using a regional climate model,* International Journal of Climatology, vol. 41, no. 1, pp. 647–662, May 2020.