UPPSALA UNIVERSITET

UPTEC STS 23021

Examensarbete 30 hp

Juni 2023

Performance Optimization of Ice Sheet Simulation Models

Examining ways to speed up simulations, enabling for upscaling with more data

Fredrika Brink

Civilingenjörsprogrammet i system i teknik och samhälle

UPPSALA UNIVERSITET

Performance Optimization of Ice Sheet Simulation Models: Examining ways to speed up simulations, enabling for upscaling with more data

Fredrika Brink

Abstract

This study aims to examine how simulation models can be performance optimized in Python. Optimized in the sense of executing faster and enabling upscaling with more data. To meet this aim, two models simulating the Greenland ice sheet are studied. The simulation of ice sheets is an important part of glaciology and climate change research. By following an iterative spiral model of software development and evolution with focus on the bottlenecks, it is possible to optimize the most time-consuming code sections. Several iterations of implementing tools and techniques suitable for Python code are performed, such as implementing libraries, changing data structures, and improving code hygiene. Once the models are optimized, the upscaling with a new dataset, called CARRA, created from observations and modelled outcomes combined, is studied. The results indicate that the most effective approach of performance optimizing is to implement the Numba library to compile critical code sections to machine code and to parallelize the simulations using Joblib. Depending on the data used and the size and granularity of the simulations, simulations between 1.5 and 3.2 times the speed are gained. When simulating CARRA data, the optimized code still results in faster simulations. However, the outcome demonstrates that differences exist between the ice sheets simulated by the dataset initially used and CARRA data. Even though the CARRA dataset yields a different glaciological result, the overall changes in the ice sheet are similar to the changes shown in the initial dataset simulations. The CARRA dataset could possibly be used for getting an overview of what is happening to the ice sheet, but not for making detailed analyses, where exact numbers are needed.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Baptiste Vandecrux Ämnesgranskare: Elisabeth Larsson

Examinator: Elísabet Andrésdóttir

Populärvetenskaplig sammanfattning

När man studerar glaciärer kan simuleringsmodeller användas för att så verklighetstroget som möjligt visualisera hur is och snö förändras över tid. Dessa modeller är ofta implementerade i datorprogram, alltså skapade med hjälp av olika programmeringsspråk, exempelvis Python. Med hjälp av dessa programmerade modeller, kan man få en uppfattning om vad som händer med glaciärerna, till exempel om de smälter, vilka temperaturer de har på olika djup samt hur mycket flytande vatten som finns i dem. För att göra dessa beräkningar behövs data som antingen samlats in genom enbart observationer eller som skapats från observationer och modellberäkningar kombinerat.

I denna studie har två sådana simuleringsmodeller studerats och utvecklats, i syfte att förbättra dem så att simuleringarna kan köras på kortare tid. Simuleringar av denna typ kan nämligen ta väldigt lång tid att göra. Modellerna utvecklades initialt av De Nationale Geologiske Undersøgelser for Danmark og Grønland (GEUS) och syftar till att simulera Grönlands glaciärer. Denna studie syftar även till att undersöka huruvida en snabbare simuleringstid kan möjliggöra att skala upp simuleringarna, det vill säga köra dem med mer data. Simuleringarna blir då mer exakta och kan omfatta större ytor. Till en början simulerades data som enbart samlats in via observationer, här kallat AWS-data (Automatic Weather Station data). När simuleringen sedan skalades upp användes en ny, större samling data kallat CARRA som är skapat av både observationer och modellberäkningar. På grund av denna skillnad anses AWS-data vara mer exakt och används därför för att validera simuleringsresultaten som CARRA-data ger upphov till.

Arbetet följde en process som syftar till att utveckla programvara genom att upprepa de fyra följande faserna: Specifikation, Implementering, Validering, och Operation. För varje iteration, alltså ett varv i samtliga faser, valdes en mindre del av den programmerade simuleringsmodellen ut, den del som mätningar visat konsumerar mest tid. Genom att fokusera på de mest tidskrävande delarna var det troligt att få mest effekt av de ansträngningar som gjordes. Varje vald del analyserades sedan och förändrades med förhoppning om att det skulle leda till en snabbare simulering. Efter förändringen utfördes tester och mätningar för att se om simuleringstiden blev snabbare och för att säkerställa att funktionaliteten var intakt. Om så var fallet behölls förändringen.

Genom studien visade sig två tillvägagångssätt vara de mest effektiva för att göra en simulering snabbare. Det ena var att ladda ned och implementera ett redan programmerat Python-bibliotek kallat Numba. Numba erbjuder en funktionalitet som gör att valda delar av kod översätts till maskinkod, kod som en dator kan läsa och förstå direkt. Detta gör att simuleringen kan köras snabbare. Det är däremot inte alltid möjligt att implementera Numba, eftersom det ibland påverkar slutresultatet. Det andra tillvägagångssättet var att parallellisera simuleringar, det vill säga att köra simuleringar

för flera platser på Grönland samtidigt. Parallellisering är möjligt eftersom datorer har flera processorer, flera hjärnor, som kan utföra uppgifter samtidigt. Den största effekten syntes när både Numba och parallellisering implementerades. Dessa implementeringar gjorde att simuleringsmodellerna kunde köras mellan 1.5 och 3.2 gånger så snabbt, beroende på vilket data som användes samt storleken och detaljrikedomen på simuleringen.

Det optimerade programmet ledde även till att simuleringar med den nya samlingen CARRA-data kunde köras snabbare. Exempelvis körde simuleringar av 6000 datarader 1.7 gånger snabbare. Syftet med att optimera för att sedan kunna skala upp simuleringen, visades därmed uppfyllt. Då resultat från simuleringar gjorda med CARRA-data respektive AWS-data jämfördes med varandra, syntes däremot skillnader. Simuleringar med CARRA-data gav inte upphov till lika stora förändringar av exempelvis smältvatten som AWS-data gjorde. Däremot kunde liknande trender ses vid samma tidpunkter, exempelvis visade simuleringarna med CARRA-data liknande temperaturförändringar vid samma tidpunkter som AWS-data, men inte i samma grad. Precis som AWS-data skulle CARRA-data alltså kunna användas för att få en överskådlig uppfattning om vad som sker med glaciärerna, men inte för att göra detaljerade analyser där exakta siffror behövs, vilket AWS-data kan. CARRA-data kan däremot möjliggöra för att simulera större ytor, eftersom mer data finns.

Acknowledgements

This thesis, the final milestone of my master's program in Sociotechnical Systems Engineering (STS) at Uppsala University, was carried out together with The Geological Survey of Denmark and Greenland (GEUS).

I would like to start by thanking my supervisor Baptiste Vandecrux at GEUS for guiding me through this project and encouraging me to challenge myself. Not only has this deepened my understanding of optimization strategies and data handling, but also given me knowledge about glaciology and the Greenland ice sheet.

I want to thank Elisabeth Larsson, my academic supervisor at Uppsala University, for supporting me and sharing important insights throughout the research.

I also want to direct a warm thank you to Siri Nordlander, my opponent during the defence, for providing me with constructive feedback that helped improve the thesis.

Finally, a special thanks to Adam Eriksson for continuously discussing with me and always supporting me.

Fredrika Brink

Copenhagen, June 2023

Table of Contents

1. Introduction 3

1.1 Goal of the Study 4

1.2 Disposition and Conventions Used 4

1.3 Terminology 5

2. Glaciological and Technical Background 6

2.1 Ice Sheet Simulations 6

2.2 The Models of this Study 7

2.2.1 The Surface Energy Balance model 8

2.2.2 The Firn Evolution model 8

2.2.3 Structure of the Code 8

2.3 Programming Concepts 9

2.3.1 Compiled and Interpreted Code 9

2.3.2 Low-level and High-level Languages 10

2.3.3 Parallelization 10

2.3.4 MATLAB and Python 11

2.4 Techniques and Tools for Increased Performance 12

2.4.1 Python Packages 12

2.4.2 A Pythonic Code Structure 14

3. Methodology 15

3.1 The Spiral model of Software Development and Evolution 15

3.1.1 Program Understanding 16

3.1.2 Specification 16

3.1.3 Implementation 16

3.1.4 Validation 17

3.1.5 Operation 17

3.2 Testing and Measuring 17

3.3 Data Preprocessing and Adaptation 18

4. Data 19

4.1 AWS data 19

4.2 CARRA data 20

5. Performance Optimization and Data Adaptation Results 20

5.1 General Specification and Main System Requirements 21

5.2 Release 1: Performance Optimizing the SEB model 21

5.2.1 Release 1: Specification 21

5.2.2 Release 1: Implementation 23

5.2.3 Release 1: Validation 24
5.3 Release 2: Performance Optimizing the SEB model 25
5.3.1 Release 2: Specification 25
5.3.2 Release 2: Implementation 26
5.3.3 Release 2: Validation 29
5.4 Release 3: Performance Optimizing the SEB model 30
5.4.1 Release 3: Specification 30
5.4.2 Release 3: Implementation 31
5.4.3 Release 3: Validation 33
5.5 The SEB model Optimization Result Summarized 33
5.5.1 Optimized vs Original Python Code Output 33
5.5.2 Problems and Issues Encountered 36
5.6 Release 4: Performance Optimizing the Firn Evolution model 41
5.6.1 Release 4: Specification 41
5.6.2 Release 4: Implementation 42
5.6.3 Release 4: Validation 43
5.7 Release 5: Adapting the SEB model to the CARRA data 45
5.7.1 Release 5: Specification 45
5.7.2 Release 5: Implementation 46
5.7.3 Release 5: Validation 47
6. Discussion 51
6.1 The Pareto Principle and Spiral model 51
6.2 Performance Optimization through Numba and Parallelization 51
6.3 Simulating for Different Data 53
7. Conclusions 54
7.1 Future studies 55
References 56

1. Introduction

The need for continuously developing already existing software will exist for as long as a software is being used [1, Ch. 9]. Requirements change, technology evolves and deficiencies and errors in the code are discovered [1, Ch. 9]. For the software to stay accurate and relevant, it must change and improve. One new such requirement can be a demand for better performance, for reduced execution time [2]. This type of requirement is often of great relevance before upscaling a simulation model by using more data. For larger simulations to be made, higher performance is needed so that the execution time is not increased [3, 4].

Performance optimization of software, meaning to speed up the execution time without changing functionality and behavior [5, Ch. 2], can be done in several ways. Changing the program's internal structure [5, Ch. 2], focusing on the most time-consuming code sections [5, Ch. 2], [6], [7, Ch. 1] and implementing specific libraries such as NumPy and pandas if coding in Python [3] are some approaches. These specific libraries allow for reusing common functions [3] and therefore not having to write repeated code. Implementing packages such as NumPy and pandas can also enable a fast way of handling data [7, Ch. 1]. Performance optimization aims to reduce expensive instructions need for resources [2].

Glaciology is a research field in which simulation models are used for studying ice sheets and surrounding climate [8]. Performing simulations is in general computationally intensive [1, Ch. 1]. This is also the case for performing ice sheet simulations, which is very time-consuming as it produces large amounts of data [9]. The Geological Survey of Denmark and Greenland (GEUS) monitors and researches the ice sheet of Greenland [10] using simulation models for investigating how the climate is changing. Two of these models will be studied in this report, namely the Surface Energy Balance model (SEB) [11] and the Firn Evolution model [12]. These models were initially coded in MATLAB but have been migrated to Python. However, the code has not fully been adapted to the Python way of programming. Libraries could be utilized more, hard coding exists, and there are many loops. This results in a software not running as time efficiently as desired. Moreover, the data used to simulate has earlier only covered some locations in Greenland. There is now a need for simulating the entire ice sheet using a new, larger dataset. This upscaling puts new requirements on the performance, requiring faster simulations. The new requirement creates a suitable case for studying performance optimization of software, which will be done in this project.

By performance optimizing the models, an upscaling of simulating the Greenland ice sheet could be made possible. Ice sheets simulation models play an important role in glaciology and overall climate research, as they simulate how the world's glaciers are changing and impacting the global climate and sea level rise [13]. Especially if considering that one of the major contributions to rising seas is in fact the mass loss from the Greenland ice sheets [14]. If both the ice sheets of Greenland and Antarctic were to melt, a sea level rise of up to 70 meters could be seen [15, 16], with the Greenland ice sheet melting in a faster phase [16]. Even a small rise in sea level affects the climate through for example coastal erosion, groundwater contamination and a bigger sensitivity for storms [15]. Further, research has shown that the central and northern parts of Greenland were warmer in the decade of 2001-2011 than they had been in a thousand years [17]. If the global warming continues, this is expected to lead to further increased mass loss [17]. Being able to study all of Greenland's ice sheets, simplifying the research on climate change, is thus an important motivation for this study of improving the two simulation models' performance.

1.1 Goal of the Study

With the increased need for faster running simulations, this study aims to perform a systematic performance analysis of the ice sheet simulation models. By finding the bottlenecks as well as possible implementations and improvements, a performance optimization of the models can be performed for faster execution times. With faster execution times, it could be possible to simulate the models with more data. Thus, the goal of the study is to examine the ways in which a large-scale simulation model can be optimized to run more time efficiently in order for a scale up to be made.

The following research questions aims to capture the goal of the study:

- How can the code of a simulation model be performance optimized? Which strategies and methods are suitable and what is the effect of using them?
- In which ways can performance optimization of code enable upscaling a large-scale simulation model?

The optimization is conducted on a laptop with four cores and eight threads (cores and threads are explained under 1.3 Terminology and 2.3 Programming Concepts). The desired outcome is that the study can contribute with useful knowledge to those performing similar software development on laptops with limited number of cores. As the main purpose of the study is to examine ways to performance optimize a simulation program, I want to stress that the focus is on the optimization and not on glaciology. As code can differ greatly, I also want to emphasize that this is an example of how performance optimization can be performed, whereas similar implementations can have other effect on other code.

1.2 Disposition and Conventions Used

The thesis is divided into seven sections, starting with this introduction and a terminology list. Then follows a background on glaciology, the simulation models, and programming characteristics. Different ways of optimizing code are presented. Chapter 3 covers the agile software development and evolution methodology used, the preprocessing and adaption of data, and the testing and measuring procedure.

Thereafter, is a chapter presenting the data, followed by the results chapter. The results are then discussed in Chapter 6. The last chapter summarizes the results and conclusions are drawn. Throughout the thesis, the following conventions are used:

- *Italic* is used for filenames.
- `Consolas` is used for program elements such as function names and variables.

1.3 Terminology

Some terminology will now be presented, both within glaciology and within programming, aiming to support the reader throughout this report.

Glaciology terms:

- Albedo – Energy from the sun reflecting back into space, is part of the energy balance [18].
- Firn – The state of snow that has not yet become glacial ice. It is the uppermost 40-120 meters of the ice sheets and is part of the accumulation zone [19]. The firn is very porous, and air can be transported through the firn's pores, exchanging air with the atmosphere, or transporting it down into the firn. The firn consists of many layers, which have different densities depending on for example the amount of air in it and the weight on top [19]. When the firn density reaches approximately 830-920 kg m^{-3} it is considered ice [19, 20].
- Glacier – Snow that has transformed into ice over many years creating a large ice mass [21].
- Heat flux – A measure of heat flow, namely the amount of heat going from a warm area to a cold area per unit area [22].
- Ice sheet – A large continental glacier. Today only two ice sheets exist, the Greenland respectively the Antarctica ice sheets. [21]
- Monin-Obhukov length – A length used in the Monin-Obhukov similarity theory describing flow and turbulence in atmospheric surfaces [23]. At height z, the Obhukov length stability parameter describes the relation of certain sources for turbulence [24]. For stable stratification the Obhukov length is positive, for unstable, it is negative [24].
- Radiation – The movement of waves or particles resulting in energy, the sun is one example of a radiation source [25].
- Roughness length – The measure of a surface's roughness [26].
- Stratification – The alternating of thick and bubbly, respectively thin and clear ice layers, as a result of different weather conditions as snow or meltwater that has refrozen [27].
- Sublimation – The transition phase from solid state to vapor state [20].
- Surface Energy Balance – Energy is constantly moving at the surface of earth, either towards the surface or away from it, depending on different physical phenomena. A balance occurs. [11] More under 2.2.1 The Surface Energy Balance model.

Programming and technical terms:

- CPU – The processor of a computer, the central processing unit, works as a brain to the computer [28]. The processor has several cores, which one can see as logical processors for performing independent tasks [29]. See 2.3.3 Parallelization for more details.
- Function – Segment of code that is called and often given information. Used for different operations, such as for plotting, receiving data, or for statistics. [3]
- Grid – A data structure that can have any shape, have multiple dimensions, and consist of multiple data arrays [30]. It is relevant in this project, as the CARRA dataset consists of gridded data [31].
- Library – Sometimes used interchangeable with package, but can also be seen as a collection of packages [32]. A library provides standardized functionality of operations that is ready made, can be imported and implemented in code [33].
- Module – A Python module contains definitions, for example of numerical and visual operations or of data structures such as arrays [33]. The modules are called from Python, but parts of them can be implemented in other programming languages as for example C [3].
- NetCDF – Network Common Data Form, is a data format often used for handling data within science [34]. It makes it possible to store large amounts of data, including information about latitude, longitude and time, which makes the format suitable for geodata [35].
- Package – Usually defined as a subset of a library, providing wrappers to already defined functions [33]. Can be viewed as a collection of modules [32].
- Reanalysis – Dataset consisting of a combination of observations and modelled outcomes, for a consistent description of the climate [36].
- Running/execution time – the time it takes to run a simulation. The time is given from subtracting the end time from the start time, using the `time` package to measure the time (more under 3.2 Testing and Measuring).

2. Glaciological and Technical Background

2.1 Ice Sheet Simulations

This chapter starts with a general description of ice sheet simulations. Within glaciology, numerical simulation models are often used [8]. Being able to simulate how the ice sheets are changing is necessary for understanding ice sheet behaviors and the contribution to sea level rise [15]. The models make it possible to analyze what has happened with the ice sheets in the past, as well as predicting the future. Ice sheet simulation models are good at simulating changes, but need to be improved to reduce uncertainties and more accurately estimate what is happening with the ice [15]. For example, the models need to have high spatial resolution [8], meaning that they should be fine and detailed. If working with gridded data, it is claimed that horizontally the grid cells should be smaller than 10 kilometers for a good result, even though a narrower area is often wished for, depending on the location and the purpose [8]. With too large grid cells, computational limitations can occur leading to wrong and underestimated results [15]. If improving the models, the meltwater from glaciers can be better simulated and the biggest uncertainty of sea level rise contribution, namely, the meltwater from Greenland and Antarctic [15], could be better understood.

A central part of modelling the ice sheets are mass balance equations, which can be applied when modelling ice volumes changes such as melting and accumulation [15]. For example, with surface mass balance equations, the interactions between ice and climate can be estimated [8]. These ice sheet simulations often rely on data, collected from for example satellite or aircraft observations, from measurements or from atmospheric modelling [15]. Acquiring enough data is important and challenging. With more observations, more reliable predictions can be made [15]. Some of these models are numerical, meaning that complex equations attempting to describe real physical processes are being solved through numerical methods [37]. The physical process is described as accurately as possible by a mathematical equation which a computer often tries to solve. The equations become complex as they aim to take into account the many complex processes happening in reality [37]. Describing reality with a model can never be made without uncertainty, as it is impossible to take every phenomenon into account or to use flawless and infinite amount of data. As with all models, one should consider the model outcomes as predictions and not as the truth [37].

2.2 The Models of this Study

The code for simulating the glaciers in this project consists of two models, the Surface Energy Balance (SEB) model and the Firn Evolution model, and can be found on GEUS' GitHub [38]. As mentioned, the code has been migrated from MATLAB to Python. The following paragraph is based on the Python code [38] and on conversations with the scientist and one of the authors of the codes, Baptiste Vandecrux (January 2023, oral communication). The models use input data containing constant values, weather data, and initial state data for a chosen weather site. When running the simulation models, layers with data about the snow, ice, and water in each individual layer are created for different time stamps. The number of layers, that is how detailed the simulation will be, can be changed. The output is saved and can be opened in other visualizing programs or be plotted directly in Python.

The difference between the two models is how they receive the surface temperature. The SEB model numerically calculates the surface temperature for each time stamp, by using a surface energy balance equation, while the Firn Evolution model uses an inputted static surface temperature. Once having the surface temperature, both models use the same code sections to calculate the ice sheet layers, for example their temperatures or density. Because of the difference in how the two models receive their surface temperature value, the SEB model is more complex and time-consuming than the Firn Evolution model.

2.2.1 The Surface Energy Balance model

The code of the SEB model is based on the Surface Energy Balance equation by van As et al. [11], given in equation (1), calculating the energy exchange at the surface of the earth. The numerical model makes it possible to compute the surface energy balance, without relying on observations limited to only one vertical level and with which data gaps might occur because of sensor failure [11]. Also, expeditions for collecting data are not necessary to the same extent. Instead, the numerical model aims to provide a reliable three-dimensional simulation without data gaps. The equation consists of incoming and outgoing shortwave radiation $S\downarrow$ and $S\uparrow$, incoming and outgoing longwave radiation $L\downarrow$ and $L\uparrow$, sensible heat fluxes H, latent heat fluxes $\lambda_S E$, and subsurface heat fluxes G. [11]

$$S\downarrow + S\uparrow + L\downarrow + L\uparrow + H + \lambda_S E + G = 0 \tag{1}$$

The glaciology and physics behind the equation and the simulation models is not further developed in this report, however some facts aim to give a brief understanding for the model concepts. If the longwave radiation is negative, it will cool the surface [11]. The sensible heat fluxes reduces the temperature differences between the surface and the atmosphere [11]. These fluxes can lead to strong winds which, through sublimation and snowdrift, affect the ice sheet.

2.2.2 The Firn Evolution model

The code of the Firn Evolution model is based on the model by Langen et al. [12, 39]. The model simulates several processes such as meltwater penetration, liquid water retention and refreezing of water, all affecting the firn (snow that has not yet become glacial ice [19]). The subsurface temperature controls how much water will freeze. The ice sheet surface is continuously changing because of melt, sublimation as well as snow and rainfall [12]. The temperature and snowpack density affects how deep the meltwater will penetrate the snow layers [12].

2.2.3 Structure of the Code

The structure of the code will now be presented. Figure 1 show the SEB model and Figure 2 the Firn Evolution model. The different files have different colors, in cursive is the file name. The difference between the two models becomes clear when comparing the figures. While the Firn Evolution model makes calls directly to *lib_subsurface.py* for updating the layers, the SEB model instead begins with iteratively calculating the surface temperature, in *lib_seb_smb_model.py*, before calling *lib_subsurface.py*. In other words, the computations SEB performs in *lib_seb_smb_model.py* makes the difference between running the two main files.

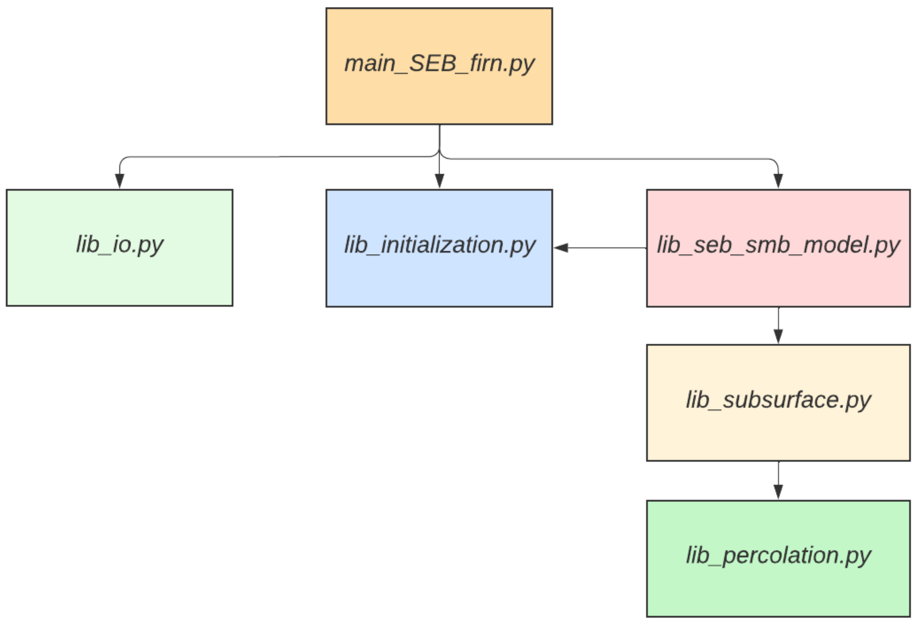


Figure 1. Code structure overview of the Surface Energy Balance model.

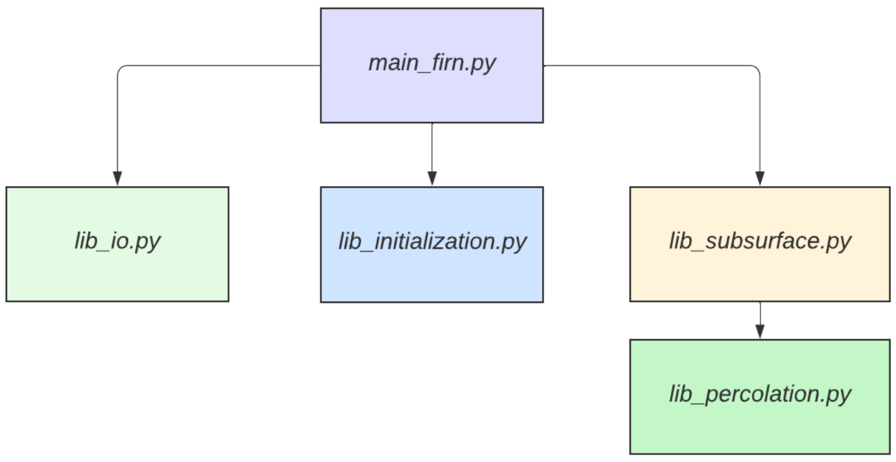


Figure 2. Code structure overview of the Firn Evolution model.

2.3 Programming Concepts

Some characteristics of programming languages as well as advantages and disadvantages of Python will now be presented.

2.3.1 Compiled and Interpreted Code

Programming languages can either be compiled or interpreted. The difference lies in how the computer gets the information and executes it. Interpreted languages execute one code line at a time, and translate each line into machine code through an interpreter, so that the computer can read and execute the code [40]. The translation of source code to machine code is done in parallel with the execution each time a program is run, making it less efficient [28]. Interpreted languages are often considered more flexible, interactive and extensible [41]. Compiled languages on the other hand, compiles the code to machine code directly and creates an executable file [40]. Compiled language usually runs faster than interpreted [40], as the source code does not have to be translated when it has already been converted into machine code.

2.3.2 Low-level and High-level Languages

Programming languages can also be of low-level or high-level, referring to its level of abstraction from the computer [28]. A low-level language can be directly read by the processor of a computer [28, 40]. The low-level code performs basic commands and is hard to read for a human. Examples of low-level languages are machine code and assembly language [28, 40]. The high-level language is written in natural language which a human can easily read. However, for a computer to read it, it must be turned into low-level language by a compiler or interpreter, as explained above. Python is an example of an interpreted, high-level language [40] executed at runtime. It is compiled through an interpreter that transforms the source code into byte code which is then interpreted by the computer [41].

2.3.3 Parallelization

Another aspect of programming is how the code is being executed. The tasks of a program can be run either sequentially, each task after the other, or at the same time, in parallel [29, Ch. 1]. This leads us to parallelization. In scientific computing, it is not unusual that simulations run for weeks, but if making different tasks parallelized the simulation can be made much faster [3]. Therefore, when multiple tasks are independent of each other, it is a great idea to make them run in parallel [42]. Parallelization is possible as computers have several processors, which in turn have several cores [29, Ch. 1]. A core is a processing unit, and each core has the ability to perform a task at a set time [43]. The cores are independent and can therefore perform their individual tasks simultaneously, meaning that they do not need to wait for another task to finish [29, Ch. 1]. A benefit from this is a gain in efficiency as computations can be made simultaneously on several computers, cores or processors, instead of on just one [3]. The processing time is reduced which can lead to improved performance [29, Ch. 1].

When discussing parallelization, there is also the concept of threads. Threads are execution lines in a process [29, Ch. 4], executed by the core and can be explained as a sequence of tasks [43]. The threads within one process share memory space [29, Ch. 4] which makes it easy to communicate between the threads [43]. If running two threads per core, instead of one, hyper-threading is enabled [44]. Hyper-threading is not the same as multiprocessing as the CPU resources are not duplicated, it just means that the core can run two threads simultaneously [44]. However, this does not always lead to twice the speed. If two threads need to access the same CPU resource, they will be run in sequence and consume more time [44]. To summarize, the processor of a computer, the CPU, controls its cores which in turn has threads that execute tasks and processes.

There are three ways of parallelizing; through shared memory parallelization, through distributed memory parallelization, or if combining these, through hybrid parallelization [3]. Källén [3] explains that shared memory parallelization is run on cores that share memory space, while the distributed memory is run on processors that has their own memory space. Parallelization on shared memory is usually executed on different threads, while the distributed memory often divides computations on different processing units [3].

By parallelizing a program one can expect it to run more efficient, as less time is spent waiting for other tasks to finish, which optimizes the performance [29, Ch. 1]. One way to estimate the performance from parallelizing is to use Amdahl's law [45]. The law includes the fraction of tasks that must be performed in sequence, the fraction that can be parallelized, the performance when running in sequence, and the number of processing elements [45]. Amdahl's law provides a theoretical performance increase, but in contrast to the Roofline model [2], it does not consider memory boundaries. The Roofline model, which is getting more and more popular, takes into account how the performance is affected by both computations and by memory accesses [2]. According to the Roofline model, this creates a limit for how high performance one can get. As the memory bandwidth, the amount of data that can be transferred per time unit [46], cannot be increased, an upper bound for the performance is set [2]. As described above, this happens when several threads try to access the shared memory at the same time. Since the bandwidth limits how many data can be delivered per unit of time, the threads need to wait for data to arrive.

Depending on whether an application's performance is bound by computations or by memory, there are different ways of acting. If an application is compute-bound, one can increase the performance through optimizing the compilation. If memory-bound, one should reduce the memory operations or eliminate memory accesses – thus reducing the number of calls to the memory [2]. Because of limitations due to bandwidth and memory accesses, it is not possible to get a perfect theoretical speed-up through parallelizing. If applying Amdahl's law, one might get a theoretical peak performance, which in reality will be lower since the application is bound by the memory bandwidth, according to the Roofline model.

2.3.4 MATLAB and Python

A comparison between programming languages will now be presented. As mentioned, the simulation models have been migrated from the programming language and numerical computing environment MATLAB [47] to the open source language Python. The main reason for moving the code was that MATLAB requires a bought license whereas Python is free to use. Having the code in Python therefore meant that the code would be available. However, there are more reasons for keeping the models in Python, even though it might not always be considered the most efficient language. Some reasons will now be presented.

Before computational scientists started using MATLAB for high-performance numerical simulations, compiled languages such as C and Fortran 77 were mainly used [33]. Once the demand for more flexible languages came, these were developed into C++ and Fortran 95 [33]. What makes the low-level languages C and C++ efficient and fast is that they compile code like a traditional computer and efficiently use the hardware [48]. However, since MATLAB could offer a simpler syntax, built-in functions and integrated simulations and visualizations, while keeping the executions fast, it became a popular language [33]. Today, these benefits can also be gained using Python, which is considered even more flexible, powerful, and convenient [4, 33, 49]. Python is considered easier to use and learn as the code is written more compact and since the syntax is simple and similar to other languages [4, 47]. When extended with libraries, Python also offers efficiently compiled code and supports parallel programming [33], as it can use other languages as foundation for computations [50, 51]. For example, many consider Python as inefficient in its handling of arrays and loops, which can be solved if implementing the package NumPy [33] (see 2.4 Techniques and Tools for Increased Performance). The integration between Python and compiled code is made with a simple interface and many tools exist to make it convenient [33]. Therefore, high-level parallel programs can be written in Python with increased efficiency [33]. If taking advantage of the many libraries, these features make Python a language with much flexibility and possibilities to create efficient software. Since Python is free, it is convenient for people to use. These features have made Python popular within scientific computing [4, 33, 49] and make it a suitable language to use for the ice sheet simulations of this study.

2.4 Techniques and Tools for Increased Performance

There are several ways for increasing the performance of a simulation, one is to use Python packages. In this study, the packages NumPy [52], Numba [53] and pandas [54] are presented as they were already partly implemented in the original code and since they are some of the most popular packages to use within scientific computing [3, 4, 33, 51]. Joblib [55] was chosen as it is claimed to be the most efficient and stable parallelization package [56] and since it works well with NumPy arrays and large objects [42]. A second approach for improving performance is to write more Pythonic code [57, Ch. 2], discussed later in this section.

2.4.1 Python Packages

The NumPy package was created for scientific computing and provide multi-dimensional array objects and fast ways to perform operations on these in compiled code [58]. An array is created as an `ndarray` object of n-dimension, with a static size initialized at creation, consisting of elements of the same data type [4, 58]. It enables for compiling code in low-level language C instead of in Python which speeds up the execution, either through (i) vectorization, by grouping operations together and computing them in C instead of going through a for-loop in Python, (ii) through reducing copying of data in memory by instead pointing to the memory for the same data, instead of entirely copying the array to the memory, or (iii) by minimizing the number of operations [59]. The `ndarray` is a data memory buffer [4] meaning that it temporarily holds data instead of having the processor immediately writing to the main memory, this enables a faster handling of data [60].

To further optimize NumPy code, the library Numba can be implemented [4]. The Numba compiler can be applied on Python code that utilizes the NumPy `ndarrays` to get an even faster execution time at native machine code speed [61]. With Numba, computationally intensive parts of code can be compiled in low-level language without having to rewrite it [4]. The user must only apply a just-in-time-decorator, a `@jit`-decorator, to the function which signals that it should be compiled at runtime. By applying the decorator, the function will be compiled to machine code [4, 61]. What happens is that Numba creates a machine code version of the function, which in the future will be the one used whenever the function is called [61]. The decorator indicates that the function object should be replaced with a special object that starts the compilation when the function is called [4]. If the exact call argument has been called before, the earlier version of the compilation will be reused, if not, it will compile that specific version [4]. Because of this, the first time a call is performed, the execution time will be longer. It is suggested that only smaller parts of code, the most performance critical ones, are compiled through Numba [62]. This can both decrease the compilation time and reduce the encountering with unsupported features. An unsupported feature is a Python specific language syntax that Numba cannot interpret [63].

The benefits of low-level languages, discussed under 2.3.2, can through compilation through Numba be gained. There are two modes in which the Numba compilation can be performed. First, Numba will try to run in a so called “no python”-mode, and if not succeeding, Numba will go back to using the “object”-mode [64]. The best performance is achieved in no python-mode, when the function can be executed without any Python interpreter involvement [61]. To ensure that the Python interpreter is not used, that the function is run in no python-mode, `nopython=True` can be added to the `@jit`-decorator, or more concisely, by simply writing `@njit`. If the `@njit`-execution is not possible, because the code cannot be executed without the Python interpreter, an error will instead be thrown [65]. The code must then be rewritten with features supported by Numba. Without `nopython=True` added to the decorator, the function might fall back into object-mode. In object-mode, only parts of the function will be compiled efficiently in Numba, and others run in the Python interpreter [61]. This is why the object-mode is less efficient. Another possibility to further increase the performance is to add `fastmath=True` to the `@jit`-decorator [64]. When running with `fastmath` the accuracy is lowered as the IEEE 754 standard, a standard for how to handle floating-point arithmetic in computer programming [66], is not strictly followed.

Furthermore, to receive an efficient data structure the pandas [54] library can be implemented. pandas offers many operations and building blocks for working with data, through for example the `DataFrame` class and its associated methods [49]. With pandas it is possible to add labels to an `ndarray`, stored as indexes, that makes it possible to work with data when objects are labeled differently [49]. The index stores the labels as both an `ndarray` and as a dictionary for value mapping [49]. The pandas library also provides an efficient way to handle datasets by implementing some parts in C [3].

As discussed under 2.3 Programming Concepts, one way to create efficient code is to parallelize the program. One reason for Python to be considered one of the slower programming languages is in fact because its default implementation is single threaded [42], meaning that no tasks are done in parallel threads. Fortunately, there are several packages for parallelizing in Python, with Joblib being one popular. Joblib can easily be implemented to divide a program's tasks across several processors or threads [42]. Not only does it enable parallelization, it can also avoid repeated work and suspend or resume jobs, in the case of a crash for example [42]. Joblib has been optimized for NumPy arrays and other large objects [42], which makes it suitable for this project.

2.4.2 A Pythonic Code Structure

The second optimization strategy will now be presented, namely, to write Pythonic code. Writing Pythonic code means to write proper code according to how the Python programming language works and to follow its formatting conventions [57, Ch. 2]. Some benefits that can be gained by writing Pythonic code are better performance, more effective executions, and code that is easier to read and understand [57, Ch. 2].

In Python it is possible to create compact expressions, so called comprehensions, which can result in faster running code [57, Ch. 2] [67]. Instead of writing several lines or loop to assign values, data structures can be created in a more concise way using only one line, often containing one Python operation [57, Ch. 2]. Through comprehensions, the amount of operation calls and time-consuming loops can often be reduced, resulting in better performance while preserving the functionality. However, it does not necessarily result in higher performance [57, Ch. 2]. Using assignment expressions, in comprehensions as well as in other code parts, can also improve the performance [57, Ch. 2]. The assignment and returning of a value are then made at once. Calling a function will be done when necessary, for example when a condition is fulfilled, with a temporary identifier assignment [57, Ch. 2]. This makes it possible to both assign a value and perform an operation on it in one line instead of several.

Furthermore, the idea of not writing repeated code will be discussed. Not having duplicate operations or statements leads to better code, and can be achieved by following something called behavioral patterns [57, Ch. 9]. Just as when importing an already created Python package for reusing code [57, Ch. 10], code written in one's own script can be reused. The Don't Repeat Yourself-principle should be followed. Reusing code can sometimes help when scaling up, as the same function can be called from more places and for different applications [68, Ch. 4]. Furthermore, one can optimize the imports to a script. By only importing the function needed from a library, and not the whole library, one might get a faster running code [67, 69, 70]. For example, if only using `DataFrame` from the pandas library in a script, one should import `DataFrame` from pandas, and not the whole pandas library. By taking advantage of different programming features, and by applying the tools and techniques described, it will be investigated whether the two ice sheet simulations model can be performance optimized. The methodology for how this was done will now be presented.

3. Methodology

The study followed a software development and evolution process, with elements of software maintenance. The main goal was to first identify the most time-consuming code sections and then to optimize these. The strategies described in 2.4 Techniques and Tools for Increased Performance were used within the process. Later, data preprocessing and adaptation was also performed.

3.1 The Spiral model of Software Development and Evolution

To enable a structured process, Sommerville's [1, Ch. 9] spiral model of software development and evolution presented in Figure 3 was followed. The model visualizes the development process including phases of specification and requirements, design and implementation, validation and testing as well as operation activities [1, Ch. 9]. Sommerville [1, Ch. 9] has adapted it for working with already existing software, which made it a suitable methodology for improving the performance of the existing SEB and Firn Evolution models. The work in the spiral is divided into releases, where one release includes all software development activities [1, Ch. 9]. In this project, one release consisted of choosing one part of the code to performance optimize. Once the changes in one release were done the next one was initialized, though sometimes different releases can be developed simultaneously [1, Ch. 9]. In addition to the activities in the spiral model an extra activity of program understanding was initially performed. This is necessary when performing software maintenance [1, Ch. 9], which the work with the models could be claimed to be. The maintenance part aims to improve and adjust a system to new requirements and correcting earlier undetected errors [1, Ch. 2]. By adapting a combination of software maintenance and the spiral model of software development and evolution, the project was conducted. The process, originated from Sommerville's [1] models, will now be presented step by step.

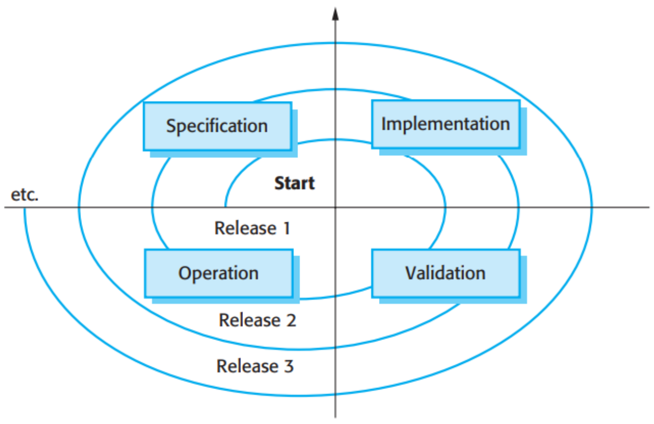


Figure 3. Sommerville's [1] spiral model of software development and evolution.

3.1.1 Program Understanding

To receive an understanding for the software, the project began with studying the existing code. This was done systematically by going through one file at a time as well as mapping how the different parts were connected. This resulted in an overview diagram (see 2.2.3 Structure of the Code). To find the most time-consuming code parts, the so called hot spots [5, Ch. 2], it was necessary to profile the program. The purpose of profiling code is to understand in which sections most time is spent and thus where to focus [71, Ch. 30] [51]. Because of this, initial performance tests were performed to find bottlenecks and hot spots (for more details, see 3.2 Testing and Measuring). By finding the most inefficient parts the focus could be put on these. This followed the Pareto Principle, a principle claiming that one can get 80 percent of the result using only 20 percent of the effort [71, Ch. 25]. It also enables finding the approximately 20 percent of the code that is responsible for 80 percent of the defects, as Boehm and Basili [6] claim is often the case. Thus, if finding and focusing on the hot spots, the chance of getting the largest improvement in execution time increases instead of working on code sections with little effect.

3.1.2 Specification

Once an understanding of the program had been gained, the spiral process could begin by specifying the requirements. Requirements in the sense of understanding how and what the software should produce [1, Ch. 1]. The main purpose of the simulation models was first studied (see 2.2 The Models of this Study). At later stages, when focusing on one time-consuming functionality at a time, that part's specification was made in more detail. With the goal of the whole project in mind, it is worth mentioning that the requirement of having a fast-running simulation was consistently relevant.

3.1.3 Implementation

The third step was to implement changes to the simulation models. When performing changes in a software evolution process, it is often an iteration of designing how the revision should look like, implementing it as well as testing it [1, Ch. 9]. When a time-consuming functionality had been identified and chosen, the execution time was measured, and a test written. Before making changes to a code section, a test for that part should be written aiming to discover if errors or functionality loss occurs as a result of the implementation [5]. Once a test was in place the earlier described tools and techniques were investigated aiming to find one suitable for the chosen functionality.

3.1.4 Validation

Once an implementation had been made, the model was again evaluated. If the implementations led to speeding up the simulations while maintaining the functionality, the changes were kept. It was validated that the model still fulfilled the user's needs and that the earlier specified requirements were satisfied [1, Ch. 8]. In this phase it was important to measure eventual impacts the changes had on the running time and if possible, identify the reason for it. More about this under 3.2 Testing and Measuring.

3.1.5 Operation

The last stage of the spiral process, the operation activity, is to put the software into use [1, Ch. 2]. If the change had a positive impact on the performance, without causing errors or changing the required functionality, the new release was pushed to GitHub. The spiral process was then repeated throughout the project.

By using the above-described methodology, the study could be performed systematically. Initially all focus was on the SEB model as it is the most complex and computationally costly one. Also, when solving and speeding up the SEB model, time-consuming parts of the Firn Evolution model could potentially also be fixed as the two models share some code sections.

3.2 Testing and Measuring

This section will cover the testing and measuring of the software and its performance. The hardware environment in which the project was conducted consisted of an Intel® Core™ 8th Gen i5-8250U processor with 8 GB RAM. The laptop had four cores and eight threads. To measure an eventual speed up, the execution time before and after the implementation was compared. Both for the system, and for the specific code section in the current release. The speed up, S, was given by equation (2), where the old execution time is divided with the new.

$$S = \frac{T_{Old}}{T_{New}} \tag{2}$$

The measuring of the CPU running time was made using the imported module `time` whereas profiling the code was made using `cProfile` [72] and `line-profiler` [73]. By running `cProfile` and studying the results, it was possible to measure the number of function calls and identify which sections were most time-consuming. Through `line-profiler` it was possible to see how much time was spent on each line of the function being profiled. The execution time was seen to depend on the computer and its status; therefore, it was standard to measure while charging the computer and with all other programs shut down. When measuring the impact of a change, the time was measured with and without that specific change at the same occasion. By doing this, it was possible to accurately see the actual impact of the change. As the conditions were as similar as possible and since new measurements of the original code were done continuously, the result is believed to be representative.

In addition to measuring eventual differences in time, tests for ensuring intact functionality were created. When improving the performance, small changes were made step by step, and after each step the code was again compiled, tested and profiled [5, Ch. 2]. If no improvement was seen, the change was made undone. The tests made sure that the in- and output were of expected type, that no errors occurred and that the purpose of the functionality remained. Tests and measurements were initially made with 6000 data rows to get fast feedback of the implementation. To see the whole impact, all data rows were used. One part of the validation was to compare 500 executions of the optimized code when using a small, static set of data input values. This did not cover all possible cases but could give a fast indicator whether some parts were wrong. To get a more detailed comparison, output files generated by the simulation model were studied and compared with outputs from the original files, when using all data. Worth noting, smaller differences between the original and optimized code were expected and considered ok as the simulation is based on numerical methods leading to smaller variations in the result. Larger differences, that were not due to random errors, were indicators for removing an implemented change.

3.3 Data Preprocessing and Adaptation

The second part of this study was to run the models with new data from the CARRA dataset. The process started with extracting the data files of the format NetCDF (see 1.3 Terminology) from GEUS's remote storage. As a lot of data existed, only data from areas close to the KAN_M and KAN_U weather sites was extracted from CARRA, to enable conducting a comparison with the AWS data. Thereafter, the preprocessing of data started, a process necessary in order to receive accurate, complete and consistent data [74, Ch. 3]. If executing a model with faulty data, the output will be faulty as well.

The following three parts were relevant when preprocessing the CARRA data: 1) Data cleaning, 2) Data transformation, and 3) Data scaling. Data cleaning is the process of finding and handling missing data or wrong, also called noisy, data [74, Ch. 3]. If the data is dirty, it can yield errors. Data with missing or wrong values might have to be removed or repaired [74, Ch. 3]. One approach to handling missing values is through data imputation, namely, to fill in the missing values [74, Ch. 4]. The data imputation can be done by estimating the missing values, based on the existing observations, and fill the empty spots with these. To get the data to match the algorithm, the process of data transformation is necessary [74, Ch. 3]. The transformation can be about changing an existing variable, for example by changing the unit, or creating a new variable from existing ones by applying mathematical formulas. The scaling of data can be seen as a sort of data transformation and is necessary to get the data in the correct scale, for example from percentages to ranging between 0 and 1. There are many more methods and processes for handling and processing data, such as dimensionality reduction, but due to time limits this thesis cannot cover them all. Once the CARRA data was preprocessed, it was necessary to adapt the scripts of the simulation models so that also the processed data could be used. The simulations were executed, and the results could be compared with the results from using AWS data.

4. Data

4.1 AWS data

In the first four releases, data collected via GEUS' Automatic Weather Stations (AWS) was used [75]. Initially, data from the weather station KAN_M was used, later also from KAN_U (see Table 1). The locations of KAN_M and KAN_U can be seen in Figure 4 below, presenting a map of Greenland and all AWS weather stations. The KAN_M dataset is smaller, with around 29 000 rows from September 2011 to January 2015, whereas KAN_U reaches from April 2009 to January 2020 and consist of around 94 000 rows. The data exists for every hour. The KAN_M site is located where it is mainly ice and only little snow, according to a conversation with scientist Baptiste Vandecrux (February 2023, oral communication). Vandecrux explains that the KAN_U site is located higher up than KAN_M, has a lot of porous snow and only little hard ice (February 2023, oral communication). If using datasets of different character, it can be possible to find out if different code parts consume most time. Using diverse datasets during the optimization is believed to enable for faster running simulations for more varieties of datasets in the future.

Table 1. Overview of which data is used in which release.

Release	Dataset	Number of rows
1	KAN_M	29 270
2	KAN_M	29 270
3	KAN_M, KAN_U	29 270, 94 323
4	KAN_M, KAN_U	29 270, 94 323

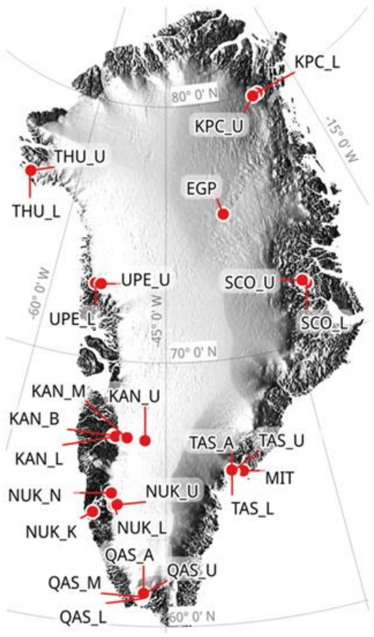


Figure 4. The AWS weather stations locations [76]. KAN_M and KAN_U in the lower middle.

4.2 CARRA data

After the optimization, the SEB model was run with the C3S Arctic Regional Reanalysis (CARRA) dataset [31]. CARRA consists of gridded data and has been collected since 1991 to present. As it is a reanalysis dataset, it is not based solely on observations, but also on a numerical weather prediction model [31]. This means that other than inputting available observations to the CARRA reanalysis, another reanalysis set, the ERA5 global reanalysis [77], has been used, together with a dataset describing surface characteristics and high-resolution satellite data [31]. The produced dataset has a horizontal resolution of 2.5 kilometers and contains 3-hourly analyses. The dataset is large, and other than Greenland it covers the Baffin Bay, Iceland, Svalbard, Scandinavia and more [31]. The CARRA data used in this study reaches from August 1990 to August 2010, due to some variables not existing for later dates.

5. Performance Optimization and Data Adaptation Results

The results from optimizing the models will now be presented. If not stated otherwise, the simulations were made with 6000 rows of weather input data and 50 layers being simulated. First, a general specification is presented. For the first three releases, focus was on speeding up the SEB model. In the fourth release, the Firn Evolution model was optimized. The last release consisted of data preprocessing of CARRA and adapting SEB to it.

5.1 General Specification and Main System Requirements

As the purpose of this study is to investigate how the simulations can be optimized to run faster, the specifications of all releases share the same main non-functional system requirement, namely, to speed up the execution time. It is also of greatest importance that the simulation keeps its functionalities, which is the second general requirement. This includes running without causing new errors or warnings. The specifications differ slightly between the releases depending on what code section they include.

5.2 Release 1: Performance Optimizing the SEB model

5.2.1 Release 1: Specification

When profiling the SEB model code, both when using KAN_M and KAN_U data, four functions are seen to be most time-consuming, all part of *lib_seb_smb_model.py*. These are presented in Table 2 and 3. Most significantly, the function `SensLatFluxes_bulk` was identified as the most time-consuming one, by far. This function also caused warnings when being executed. Considering this, `SensLatFluxes_bulk`, was chosen for the first release.

Table 2 and 3 (and following profiling tables) consists of five columns and present how much time is spent in each function. "Number of calls" presents how many times the function was called in the execution, and "Total time" the time spent in the actual function, meaning that calls to other functions are excluded. "Per call (tot.)" is the total time for one call, in other words the total time divided by number of calls. "Cumulative time" is the time spent in the function and its' sub-functions, meaning that eventual calls to other functions are included. The "Function" columns present the function name.

Table 2. Result from initial profiling of original code. 6000 rows, KAN_M dataset.

Number of calls	Total time (in sec)	Per call (tot.) (in sec)	Cumulative time (in sec)	Function
242 887	24,57	0,00	37,77	`SensLatFluxes_bulk`
1 335 271	13,08	0,00	13,08	`SmoothSurf`
1	1,47	1,47	58,43	`HHsubsurf`
242 788	1,17	0,00	1,17	`SurfEnergyBudget`

Table 3. Result from initial profiling of original code. 6000 rows, KAN_U dataset.

Number of calls	Total time (in sec)	Per call (tot.) (in sec)	Cumulative time (in sec)	Function
242 916	25,51	0,00	38,97	`SensLatFluxes_bulk`
1 354 753	13,35	0,00	13,35	`SmoothSurf`
1	1,92	1,92	58,05	`HHsubsurf`
242 916	1,17	0,00	1,27	`SurfEnergyBudget`

Besides the general requirements, the Release 1 specification was to maintain the `SensLatFluxes_bulk` functionality. The function's main purpose is to calculate the sensible heat fluxes (`SHF`), the latent heat fluxes (`LHF`) and the Monin-Obhukov length (`L`). The function should correct for atmospheric stability, computing the values of temperature (`theta_2m`), humidity (`q_2m`), and wind speed (`ws_10m`) [38]. The "`_2m`" indicates that out of two measurement levels on the weather station, the measuring was done at the higher level, approximately at 2 meters height, where the risk of the measurement instrument getting buried in snow is lower (Baptiste Vandecrux, March 2023, oral communication). "`_10m`" indicates that the wind speed was measured at approximately 10 meters height, which is the meteorological standard (Baptiste Vandecrux, April 2023, oral communication). The function also outputs the roughness Reynolds number (`Re`), which is being calculated when calling the functions `SmoothSurf` or `RoughSurf`. If the wind speed is too low, below the limit defined in the input constants, none of these mentioned values are calculated, instead they are assigned standard values, 0, -999 or 1e-10, as the fluxes are anyhow very low then [38].

The steps of `SensLatFluxes_bulk` will now be explained briefly. The original code section of the function is long and consists of many calculations and calls to `SmoothSurf` when the snow thickness is greater than 0, and `RoughSurf` otherwise [38]. `SmoothSurf` and `RoughSurf` are called for receiving the roughness lengths, `z_h` and `z_q`, a variable called friction velocity, `u_star`, and, as mentioned, `Re` [11]. Also, `SensLatFluxes_bulk` computes the value for the saturation vapor pressure at surface, `es_ice_surf`, a parameter which is later used when computing the surface humidity, `q_surf` [38]. Following is a code section computing either stable or unstable stratification, depending on the temperature, given that the wind speed is larger than a set limit. If-statements are used for checking these conditions. In both stable and unstable stratification, stability corrections [11] are computed, but in different ways. In the code, these are named `psi_m1` and `psi_m2` for stability correction of momentum, `psi_h1` and `psi_h2` for heat, respectively `psi_q` and `psi_q2` for humidity. The roughness length scales for snow or ice are then updated, by calling `SmoothSurf` or `RoughSurf`, using the newly computed `psi_m1` and `psi_m2`. Other parameters, such as `th_star`, `q_star`, `SHF`, `LHF`, `theta_2m`, `q_2m` and `ws_10m`, are calculated and updated with new values from the different computations through several iterations.

It is not necessary to understand all the variables or computations that are being made in the function. However, a takeaway should be that the function is long, has loops containing conditional statements, and makes calls to the functions `SmoothSurf` or `RoughSurf` several times when going through for-loops. It returns the Monin-Obhukov length, the sensible and latent heat fluxes, the temperature, humidity and windspeed as well as the Reynolds number [38]. Worth noting, the function `SensLatFluxes_bulk` itself is called at maximum 60 times for each input weather data row, if balance has not been found before the sixtieth iteration. For example, if using 6000 data rows, calls to `SensLatFluxes_bulk` will be made for each data row until the energy budget (calculated in another function) has been found for that iteration, at most 60 times. Thus, considering this example of 6000 data rows, `SensLatFluxes_bulk` can be called up to 360 000 times.

5.2.2 Release 1: Implementation

The first warning to address, was a future warning informing that the pandas' method to append a `DataFrame` was to be removed from the library in future updates. `append` was not applied in `SensLatFluxes_bulk` but in the file *lib_initialization.py*. However, this warning was still addressed to enable a smooth execution of the code. Instead of the `append` function, the pandas' `concat` method was implemented for joining several `DataFrames` together. Furthermore, a runtime warning occurred when executing the code. It came from `SensLatFluxes_bulk`, when computing the saturation vapor pressure at surface, `es_ice_surf`, and was due to double scalar overflow encountered. The values overwent the limit for allowed values when using NumPy floats of the type `float64`. To resolve the error, an attempt to move the code section to its own function was implemented, in which Numba was used for compilation. It was initially believed that this solved the overtime error, while it also led to reducing the execution time by approximately 2 percent. However, it was later found that an issue that already existed in the code (discussed under 5.5.2 Problems and Issues Encountered), randomly occurred and made the simulation crash when compiling this part through Numba. The implementation was therefore not kept, and it was noted that this issue needed to be investigated further in future studies.

After attempting to solve the warnings, the requirement of speeding up the model continued. To find the most time-consuming parts of `SensLatFluxes_bulk`, when using the KAN_M data, a profiling of all the function's lines was performed using `line_profiler`. The many calls to the function `SmoothSurf` were seen to be the most time-consuming. The stability correction functions for momentum, heat, and humidity (`psi_m1`, `psi_m2`, `psi_h1`, `psi_h2`, `psi_q` and `psi_q2`) were also significant time-consumers. Because of this result, the stability correction functions and `SmoothSurf` was further studied. At first, the stability correction functions were removed from `SensLatFluxes_bulk` and implemented in one separate Numba `@jit`-decorator function. This had a major effect on the execution time (further discussed under 5.2.3 Release 1: Validation). Thereafter, changes to `SmoothSurf` were made. Namely, implementing the computation of `u_star` in a separate `@jit`-decorated function. This sped up the execution by approximately one second. When also moving the computation of `z_h` and `z_q` from `SmoothSurf` to a separate `@jit`-decorated function, a big change was noticed. The code executed 10.6 seconds faster on average. When comparing the optimized code to the original code a total speed up of 26.1 seconds had been made, a time reduction of 49 percent. However, as discussed in below validation, this was not the final version of this release.

5.2.3 Release 1: Validation

First and foremost, the change made to eliminate the future warning had desired effect, even though the runtime warning of overflow could not be solved. As this error was also present in the original code, it was decided being out of the project scope, and focus was instead put on the optimization. There were also another error occurring randomly, both in the original and the optimized code. One reason for it to only occur randomly could be due to some parts occasionally being executed in different order, leading to faults. It could also be due to the script not being able to handle some values in the data. This is discussed further under 5.5.2 Problems and Issues Encountered.

When validating the implementations, it was noticed that the results given by `SensLatFluxes_bulk` differed from the original program. More specifically, the output values of `L`, `LHF`, `SHF`, and `Re` differed from the original functions', even though the same input was used. The difference was bigger than just due to numerical iterations and the changes were therefore studied. It was identified that the changed result originated from moving the stability correction computations, the `psi`-values, in stable stratification to one Numba `@jit`-decorated function.

A possible reason for this could be due to the dependency between the variables and how they are connected. The Monin-Obhukov length, `L`, and the roughness lengths, `z_h` and `z_q`, among others, are part of the calculations of the `psi`-values in stable stratification. Thereafter, new values for `z_h`, `z_q` and `Re` are computed using the values of `psi_m1` and `psi_m2`. Then, `psi_h2`, `psi_h1`, `psi_q2` and `psi_q` are used for computing `th_star` and `q_star`, which then are used to update the values of `L`, `SHF` and `LHF`. These steps are looped through up to 20 times, which is the limit for the iteration. This explanation may be confusing, but the takeaway should be that several operations with dependencies to each other are executed, which probably leads to wrong results. When compiling through Numba, the dependencies between the variables seem to be confused or lost. Another possibility could be that Numba tries to compile all `psi`-values using the same compiled function, even though the functions are slightly different – but keep in mind, these are only speculative thoughts.

Considering the faulty outputs and great speed ups the Numba implementation led to, it was tested to instead compute the `psi`-values in separated Numba functions. This did not speed up the code to the same extent, however it led to speed ups without affecting the output. The two different ways of computing the `psi`-values were tested, and the results are presented in Table 4. The simulation ran on average 4 seconds slower when having separate functions instead of only one, a performance decrease of approximately 13.8%, but as it preserved the functionality it was considered a better alternative. These outcomes demonstrates that instead of having one Numba function, it can be necessary to split up code into smaller sections when implementing Numba, as discussed earlier (2.4.1 Python Packages). However, it does not align with the statement that splitting up functions always result in decreased execution time, since having all `psi`-values computed in one function was faster than when split up. Instead, it shows the complexity and difference between code and that the effects may not always be the same but depending on the specific code.

Table 4. Execution CPU time, different `psi`-computations (mean $\pm$ standard deviation). 6000 rows, KAN_M.

One `@jit`-decorated function (loss of functionality) (in sec)	Separate `@jit`-decorated functions (functionality intact) (in sec)
$29{,}0\ \pm 1{,}4$	$33{,}3\ \pm 1{,}5$

After moving the `psi`-values operations to separate `@jit`-decorated functions, all implementations were tested and approved. It is worth mentioning that the update of the pandas' method `append` to `concat` was made to the old code as well, for a smooth testing. When comparing execution times, the optimized code ran 1.6 times faster on average, resulting in a 20 second faster execution (see Table 5). Release 1 did thus succeed in maintaining the functionality and speed up the code.

Table 5. Execution CPU time, original versus optimized code (mean $\pm$ standard deviation). 6000 rows, KAN_M.

Original code (in sec)	Optimized code, Release 1 (in sec)
$53{,}3\ \pm 4{,}5$	$33{,}3\ \pm 1{,}5$

5.3 Release 2: Performance Optimizing the SEB model

5.3.1 Release 2: Specification

Release 2 started with profiling the code, see Table 6. `SensLatFluxes_bulk` together with `SmoothSurf` were still most time-consuming. A new function was present, the `get_zh_zq`, which was implemented with a `@jit`-decorator in Release 1. The function is called from `SmoothSurf`, computing the roughness lengths `z_h` and `z_q`. Even though this new function consumes time, the overall time in the functions were reduced after Release 1. The time spent in `SmoothSurf` in the initial profiling was 13.08 seconds whereas in this only 5.78 seconds and 1.85 seconds in `get_zh_zq`, meaning a speed-up of 1.7 times had been made to `SmoothSurf`.

Table 6. Result from profiling optimized code before Release 2, with KAN_M dataset.

Number of calls	Total time (in sec)	Per call (tot.) (in sec)	Cumulative time (in sec)	Function
243 969	20,73	0,00	33,99	`SensLatFluxes_bulk`
1 341 902	5,78	0,00	9,64	`SmoothSurf`
1 341 902	1,85	0,00	1,85	`get_zh_zq`
1	1,65	1,65	52,43	`HHsubsurf`
243 969	1,23	0,00	1,36	`SurfEnergyBudget`

A profiling of `SensLatFluxes_bulk` showed that the major execution time was still spent calling `SmoothSurf`. The several computations for receiving constants before calculating the `psi`-values as well as calculating them, when the stratification is *unstable*, were also computationally intensive. In Release 1 the `psi`-values for stable stratification were implemented as `@jit`-decoratored functions, but the values for unstable stratification remained as in the original code. Moreover, the computations of the parameter `th_star`, `q_star` as well as `SHF` and `LHF` also consumed time. These computations consisted of multiplication, division, and subtraction with objects of types `float`, `numpy.float64` and `int` [38], see Code block 1.

```
th_star = c.kappa * (theta - Tsurf) / (np.log(z_T/z_h) – psi_h2 + psi_h1)
q_star = c.kappa * (q - q_surf) / (np.log(z_RH/z_q) – psi_q2 + psi_q)
SHF = rho_atm * c.c_pd * u_star * th_star
LHF = rho_atm * c.L_sub * u_star * q_star
```

Code block 1. Original code example of time-consuming computations.

Considering the profiling results, the focus remained on the above-mentioned parts of `SensLatFluxes_bulk` and `SmoothSurf`. The requirements were therefore the same as in Release 1; to speed up execution time as well as to retain the functionality.

5.3.2 Release 2: Implementation

The implementation phase started with investigating the computations of `psi`-values for unstable stratification. For computing these, several constants (`x1`, `x2`, `y1`, `y2`, `yq` and `yq2`) are first prepared through mathematical operations (subtraction, multiplication, division, and exponentiation). Then, the `psi`-values are computed using these constants, no other variables are involved. Because of this, the unstable stratification `psi`-values were implemented in one `@jit`-decoratored function. It was not believed being necessary to create separate functions for them, like for the `psi`-values in stable stratification, since no dependent variables are used within the rather simple calculations. This led to the simulation executing 0.8 seconds faster on average.

Similarly, the computation of `x1`, `x2`, `y1`, `y2`, `yq` and `yq2` were put into a Numba function, which made the code execute around 1.1 seconds faster. In total, these two implementations had little impact and made the code run 1.1 times faster on average.

Considering the specification, the computations of `th_star`, `q_star`, `SHF` and `LHF` were then tried being compiled through `@jit`-decoratored functions. This led to around 1.1 times faster executions. Instead of the code in Code block 1, the following in Code block 2 was used for calling the functions. The Numba compiled functions being called from Code block 2 is presented in Code block 3.

```
th_star, q_star = get_th_star_q_star(
    c.kappa, theta, Tsurf, z_T, z_h,
    psi_h2, psi_h1, q, q_surf, z_RH,
    z_q, psi_q2, psi_q
)
SHF, LHF = get_SHF_LHF(
    rho_atm, u_star, th_star, q_star, c.c_pd, c.L_sub
)
```

Code block 2. Optimized code where computations have been moved to Numba compiled functions.

```
@jit(nopython=True)
def get_th_star_q_star(kappa, theta, Tsurf, z_T, z_h, psi_h2, psi_h1, q,
q_surf, z_RH, z_q, psi_q2, psi_q):
    th_star = (kappa * (theta - Tsurf) /
               (np.log(z_T / z_h) - psi_h2 + psi_h1))
    q_star = (kappa * (q - q_surf) /
              (np.log(z_RH / z_q) - psi_q2 + psi_q))
    return th_star, q_star

@jit(nopython=True)
def get_SHF_LHF(rho_atm, u_star, th_star, q_star, c_pd, L_sub):
    SHF = rho_atm * c_pd * u_star * th_star
    LHF = rho_atm * L_sub * u_star * q_star
    return SHF, LHF
```

Code block 3. Optimized code, the Numba compiled functions called from Code block 2.

When comparing the profiling results from before and after the change, it was noted that the calculation of `th_star`, `q_star`, `SHF` and `LHF` took around 8 microseconds per hit before the change, and approximately 4.5 microseconds per hit afterwards, namely, 1.8 times the speed. The same computations were done in another code section. Instead of having the same operations repeated twice, the code structure could be improved by calling the earlier implemented Numba functions from this part as well. The Don’t Repeat Yourself-principle was thus applied, and repetition of code was avoided while compiling more code in low-level language. This resulted in 1.1 times faster runs. When also implementing a `@jit`-decoratored function for computing `L`, which was done in stable and unstable stratification, a speed up of circa 1.1 times the speed was gained. In total, these implementations led to 1.2 times faster executions on average.

When continuing the attempts to speed up, light was again shed on `SmoothSurf`. Even after parts had been moved to `@jit`-decoratored functions, it still consumed time. A line-by-line profiling of `SmoothSurf` was therefore conducted. A part of the function had not yet been investigated, namely the part in which the value of an indicator is decided based on the value of `Re`. The indicator, `ind`, is assigned 0, 1 or 2, and passed to the function computing `z_h` and `z_q` to indicate which of three constants should be used. Three approaches to assign `ind` were examined. The first was to keep the original code, as in Code block 4. The second attempt was to comprehend the expression as in Code block 5, however, the printing had to be put outside. Lastly, creating a `@jit`-decorated function called from `SmoothSurf` as in Code block 6 and 7 was tried.

```
Re = u_star * z_0 / nu
    if Re <= 0.135:
        ind = 0
    elif (Re > 0.135) & (Re < 2.5):
        ind = 1
    elif Re >= 2.5:
        ind = 2
    else:
        ind = float("nan")
        print("ERROR", Re, u_star, z_WS, psi_m1, psi_m2, nu, ind)
```

Code block 4. Original code of assigning `ind`.

```
ind = 0 if Re <= 0.135 else 1 if (0.135 < Re < 2.5) else 2 if Re >= 2.5
else float("nan")
if ind == float("nan"):
    print("ERROR", Re, u_star, z_WS, psi_m1, psi_m2, nu, ind)
```

Code block 5. Modified code, comprehend assignment of `ind`.

```
ind = get_ind(Re, u_star, nu, z_WS, psi_m1, psi_m2)
```

Code block 6. Modified code, call `@jit`-function for value of `ind`.

```
@jit(nopython=True)
def get _ind(u_star, z_0, nu, z_WS, psi_m1, psi_m2):
    if Re <= 0.135:
        ind = 0
    elif (Re > 0.135) & (Re < 2.5):
        ind = 1
    elif Re >= 2.5:
        ind = 2
    else:
        ind = np.nan
        print("ERROR", Re, u_star, z_WS, psi_m1, psi_m2, nu, ind)
    return ind
```

Code block 7. Modified code, `@jit`-function assigning `ind`, called from Code block 6.

When profiling the different versions line-by-line, the second and third attempts were seen to use 0.6 respectively 0.7 seconds less time than the first version. However, when running the whole simulation using the three different versions, the first attempt consumed the least time and was circa 1.03 times faster than attempt two and 1.06 times faster than attempt three. The reason for it to be faster could be because the code did not contain many mathematical computations or loops, but simple if-statements and assignments. One could therefore believe that very simple and short operations, with only few mathematical operations, are not always the most suitable ones to compile with Numba. The call to the function will instead consume time. The reason for the comprehension attempt, the second one, to consume more time than the first attempt could be because of the printing in the conditional if-statement. This if-statement, aiming to print variable information when `ind` is not assigned a number is checked for each value, which consumes time. This implementation of Numba did not speed up the execution time in the way that has been seen earlier. This is a finding worth remembering in the future. The original version of the code was kept.

5.3.3 Release 2: Validation

The Release 2 implementations were validated according to the tests (See 3.2 Testing and Measuring). The outputs were the same and the functionality retained. When comparing the execution time with the results from Release 1, the execution time was 1.2 times faster on average (see Table 7), and 2.0 times faster than the original code.

Table 7. Execution CPU time, original versus optimized code (mean ± standard deviation). 6000 rows, KAN_M dataset.

Original code (in sec)	Optimized code, Release 1 (in sec)	Optimized code, Release 2 (in sec)
53,3 ± 4,5	33,3 ± 1,5	27,3 ± 1,9

5.4 Release 3: Performance Optimizing the SEB model

5.4.1 Release 3: Specification

For Release 3, the requirements from 5.1 General Specification and Main Requirements remained, and KAN_U data was to be used. To be able to measure eventual speed up in Release 3, it was necessary to first document the execution time and the most time-consuming code sections when using KAN_U data. The average execution time was 30.6 seconds for the optimized code, and 58.4 seconds for the original code. When profiling the model using KAN_M data (see Table 8) and then KAN_U data (see Table 9), the same result was given; the optimization done in Release 1 and 2 had impacted the execution time positively when using another dataset as well. The effects from Release 1 and 2 could thus be shown and a comparison between using KAN_M and KAN_U data was enabled. It was seen that the same four functions were most time-consuming, regardless of data source. Similarly, line-by-line profiling showed that the same parts of `SensLatFluxes_bulk` consumed the most time. The execution times were almost the same, simulating for KAN_M was just slightly faster. Before, it had been believed that these results could differ when simulating for different data as the conditions at the sites are different (Baptiste Vandecrux, February 2023, oral communication), but this was not the case.

Table 8. Result from profiling optimized code before Release 3.
6000 rows, KAN_M dataset.

Number of calls	Total time (in sec)	Per call (tot.) (in sec)	Cumulative time (in sec)	Function
242 887	19,22	0,00	35,13	`SensLatFluxes_bulk`
1 335 724	5,86	0,00	9,54	`SmoothSurf`
1 335 724	1,82	0,00	1,82	`get_zh_zq`
1	1,71	1,71	53,75	`HHsubsurf`
242 887	1,28	0,00	1,42	`SurfEnergyBudget`

Table 9. Result from profiling optimized code before Release 3.
6000 rows, KAN_U dataset.

Number of calls	Total time (in sec)	Per call (tot.) (in sec)	Cumulative time (in sec)	Function
243 951	19,50	0,00	35,21	`SensLatFluxes_bulk`
1 361 223	6,08	0,00	9,34	`SmoothSurf`
1	2,15	2,15	54,37	`HHsubsurf`
1 361 223	2,01	0,00	2,01	`get_zh_zq`
243 951	1,3	0,00	1,44	`SurfEnergyBudget`

5.4.2 Release 3: Implementation

When simulating the KAN_U data not all output files contained information. When studying this, it was found that the KAN_U dataset contained data rows with missing values. These 21 rows were removed from the dataset, and the simulation ran smoothly. The implementation phase then began with examining attempts to speed up the code without taking advantage of Numba. So far, it was only the Numba compilations that had led to speed ups. Therefore, it was considered interesting to study other approaches. As the calls to `SmoothSurf` were seen to consume approximately 33% of `SensLatFluxes_bulk`’s time, the effect of changing the datatype of the `psi`-values being sent to `SmoothSurf` was investigated. The `psi`-values were turned into Python `floats` when being recalculated, and to stop this from happening they were reassigned to `np.float64` to ensure them being of NumPy floats. This meant that `SmoothSurf`, and related `@jit`-decorated functions, could perform the calculations on objects of the datatype `np.float64`’s. This did not influence the time, it rather took a little bit more time, likely due to repeatedly assigning the `psi`-values to `np.float64` for each data row. Changing the datatype to `np.float64` was therefore not kept.

Thereafter, the NumPy method `clip` was implemented inside `SmoothSurf` to replace two if-statements that assign `z_h` and `z_q` to 1e-6, if they are smaller than 1e-6. The `clip` method creates a lower limit for a variable [78]. Instead of speeding up the code, the implementation led to a much slower execution time, approximately around 4.5 times the original time, and consumed 137 seconds in total. A likely reason for this is that the `clip` method was applied to all values, whereas the if-statement only changed the values if they were smaller than 1e-6. These conditional statements only change values on those small enough, not all. In this case, implementing a NumPy method had a negative impact, and the original code was kept.

Furthermore, the imports in the scripts were attempted to be optimized. Instead of importing a whole library, only the functions being used were imported. This was done in all scripts, except for the import of NumPy in *lib_seb_smb_model.py* as the number of NumPy methods were many. However, this did not affect the execution time. Some of the imports were kept in the new way, as it made the code more understandable, but the imports of pandas and NumPy were written as originally to clearly show when these libraries were used. The next attempt was to replace a several lines if-else statement (Code block 8) with a one-line comprehension (Code block 9). Comprehensions, most often list comprehensions, are often suggested to speed up Python code [57, Ch. 2] [67], as discussed under 2.4.2 A Pythonic Code Structure. In this case, it had no effect on execution time, but was kept as it was considered more readable.

```
if snowthick > 0:
    z_h, z_q, u_star, Re = SmoothSurf_opt(
        WS, z_0, psi_m1, psi_m2, nu, z_WS, c
    )
else:
    z_h, z_q, u_star, Re = RoughSurf(
        WS, z_0, psi_m1, psi_m2, nu, z_WS, c
    )
```

Code block 8. Before change: if-else statement for calling either SmoothSurf or RoughSurf.

```
(z_h, z_q, u_star, Re) = (
    SmoothSurf_opt(WS, z_0, psi_m1, psi_m2, nu, z_WS, c)
    if snowthick > 0
    else RoughSurf(WS, z_0, psi_m1, psi_m2, nu, z_WS, c)
)
```

Code block 9. After change: if-else statement in line comprehension, for calling either SmoothSurf or RoughSurf.

After the above changes were examined, with no, little, or opposite effect, Numba was again turned to. More specifically, a code section that appeared twice, in both stable and unstable stratification in `SensLatFluxes_bulk`, were put into a `@jit`-decorated function. When compiling the code section through Numba, a slightly faster execution time was seen. By again following the Don’t Repeat Yourself-principle, the maintainability of the code was also improved. Thereafter, adding `fastmath=True` to the `@jit`-decorated functions was investigated, as it can speed up an execution by lowering the accuracy of floating-point math operations. It was not possible to enable `fastmath` on all `@jit`-decorated functions, as the lowered accuracy sometimes led to wrong outputs. When only keeping `fastmath=True` where it did not affect the outcome, no effect on the execution time was seen. It rather led to slightly slower executions, and possibly less accurate computations, and was therefore not kept.

5.4.3 Release 3: Validation

The examined implementations in Release 3 had little or no effect on the execution time. Several attempts to change datatypes, change code structures and only import the used methods were made without desired effect. Similarly, the use of the NumPy method `clip` and to run the `@jit`-decorated functions with `fastmath=True` affected the outcome negatively or led to no speed-up and was therefore not kept. The implementation that did have an effect was moving two repeated code sections to a `@jit`-decorated function. The measured speed up is presented in Table 10, where the original code and the Release 2 and 3 codes are compared. An approximate 1.1 times faster execution was seen compared to the Release 2 code, and a 2.1 times faster execution compared to the original code. The same comparison was made when using KAN_M for simulating, see Table 11. A speed up was still visible; the optimized code was on average 2.1 times faster than the original code and 1.1 times faster than the Release 2 code.

Table 10. Execution CPU time, original versus optimized code (mean $\pm$ standard deviation). 6000 rows, KAN_U dataset.

Original code (in sec)	Optimized code, after Release 2 (in sec)	Optimized code, after Release 3 (in sec)
$58,4 \pm 0,7$	$30,6 \pm 0,3$	$27,6 \pm 0,2$

Table 11. Execution CPU time, original versus optimized code (mean $\pm$ standard deviation). 6000 rows, KAN_M dataset.

Original code (in sec)	Optimized code, Release 1 (in sec)	Optimized code, Release 2 (in sec)	Optimized code, Release 3 (in sec)
$53,3 \pm 4,5$	$33,3 \pm 1,5$	$27,3 \pm 1,9$	$25,6 \pm 0,3$

5.5 The SEB model Optimization Result Summarized

5.5.1 Optimized vs Original Python Code Output

After three releases, the optimization of SEB performed so far was studied visually. At this stage, the optimized code ran around twice as fast as the original one when simulating 6000 data rows with 50 layers. When studying the outputs from using all KAN_U data, see Figure 5, the graphs looked very similar with only very small differences. These differences were expected and not considered a problem as they were a result of numerical iterations.

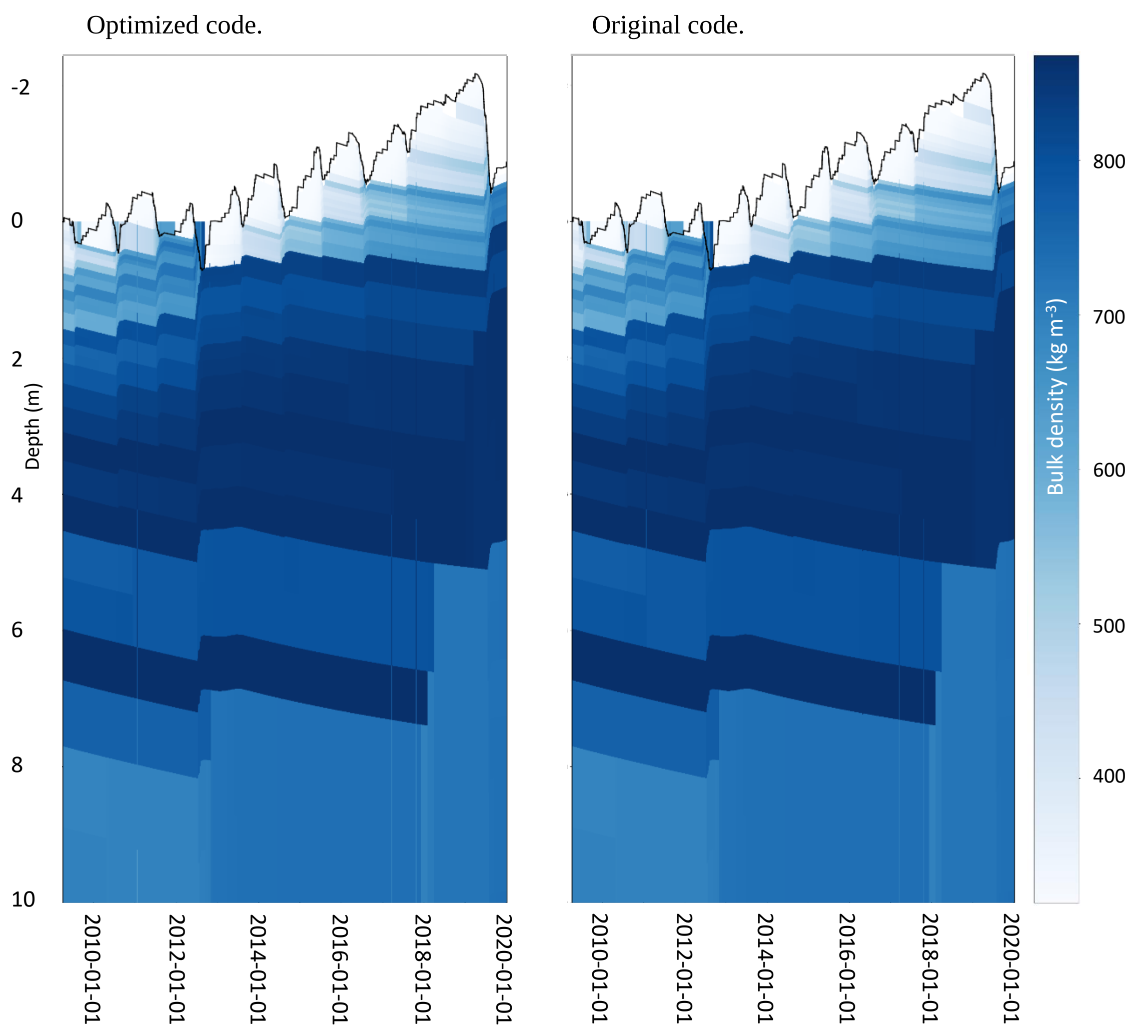


Figure 5. Left: Density output from optimized code, 50 layers. Right: Density output from original code, 50 layers.

As the simulation can be made with more than 50 layers for increased granularity, the impact of the optimization when running with 200 layers was examined as well, see Figure 6. Like simulating with 50 layers, the original and optimized code outputs were very similar but the small differences due to numerical iterations were slightly bigger. One such difference can be seen in Figure 6, in January 2020 around 0 m depth, where the original code showed a higher density than the optimized. With increased granularity, there were thus more small differences. The optimized code ran around 1.9 times faster than the original code, when simulating 200 layers with all KAN_U data.

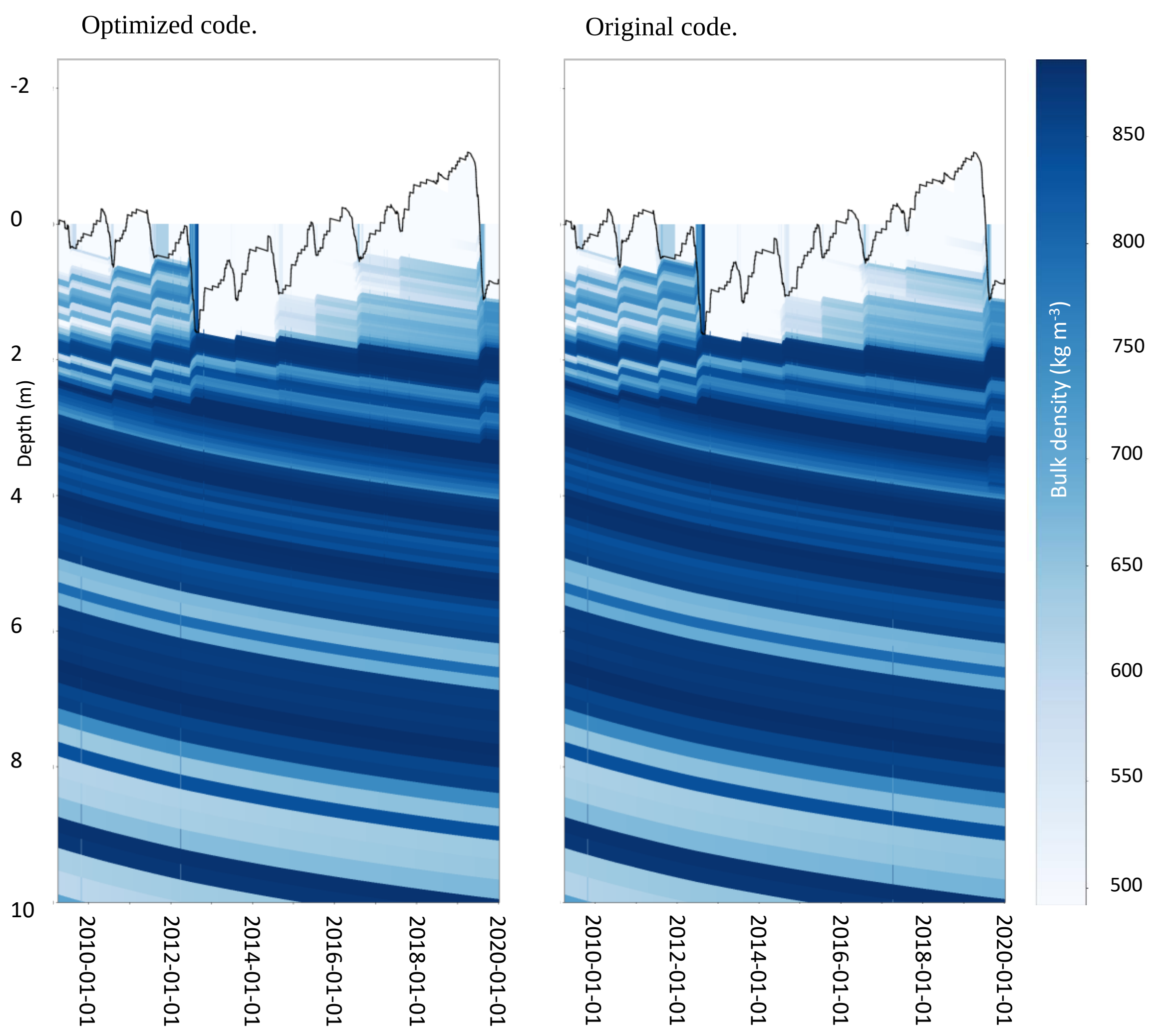


Figure 6. Left: Density output from optimized code, 200 layers. Right: Density output from original code, 200 layers.

An intermediate result from the simulation was also investigated, namely, the output from `SensLatFluxes_bulk`, the variables `L`, `LHF`, `SHF`, `theta_2m`, `q_2m` and `ws_10m`. When comparing the outputs, the graphs were very similar except for the ones plotting the `L` value, the Monin-Obukhov length, see Figure 7. It was found that `L` in rare occasions reaches to very high values, of sizes up to 40 million. This finding will be discussed below in section 5.5.2 Problems and Issues Encountered.

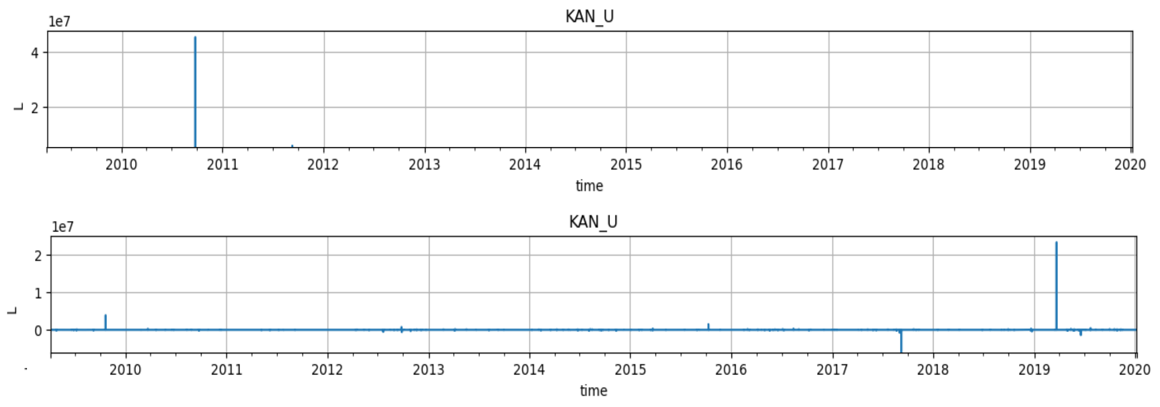


Figure 7. Above: Monin-Obukhov length, `L`*, computed by optimized code, 200 layers. Below: Monin-Obukhov length,* `L`*, computed by original code, 200 layers.*

5.5.2 Problems and Issues Encountered

The Monin-Obukhov length, `L`, was studied further by simulating 17 000 KAN_U data rows (to include the peak in the end of 2010 seen in Figure 7), limiting `L`'s y-axis between -2000 and 6000. When studying the graphs, see Figure 8, they look similar with only smaller differences. The spikey values earlier seen do not seem to affect these values. Therefore, it is believed that the spikey `L`-values in Figure 7 might be a result of numerical instabilities. Since `LHF` and `SHF` are calculated using `L`, and these were seen to not be affected, the result was considered ok, but the problem was further studied.

To understand what was happening with `L`, several simulations were made comparing the values from `SensLatFluxes_bulk`. The differences in `L` varied in size, and sometimes got as big as to a size of 4 000 000. However, these differences both occurred when comparing outputs from the optimized code with the original code, the original with the original, and the optimized with the optimized. This led to the conclusion that the differing values did not occur because of the optimization, but already existed in the original Python code. Differences of big sizes only occurred for a small part of the outputs, whereas for the majority no big differences existed, and the separate runs converged to similar values. This could be the reason for why the differences are not shown in the end result, but only in an intermediate output. A possible explanation for these large differences of `L` to only occur for some time stamps, could be due to an overflow occurring in the computations. An overflow in the sense of values getting larger than the allowed limits for that data type. As this project focuses on examining ways to speed up executions and not changing the functionality of algorithms, the issue was communicated and documented for future projects.

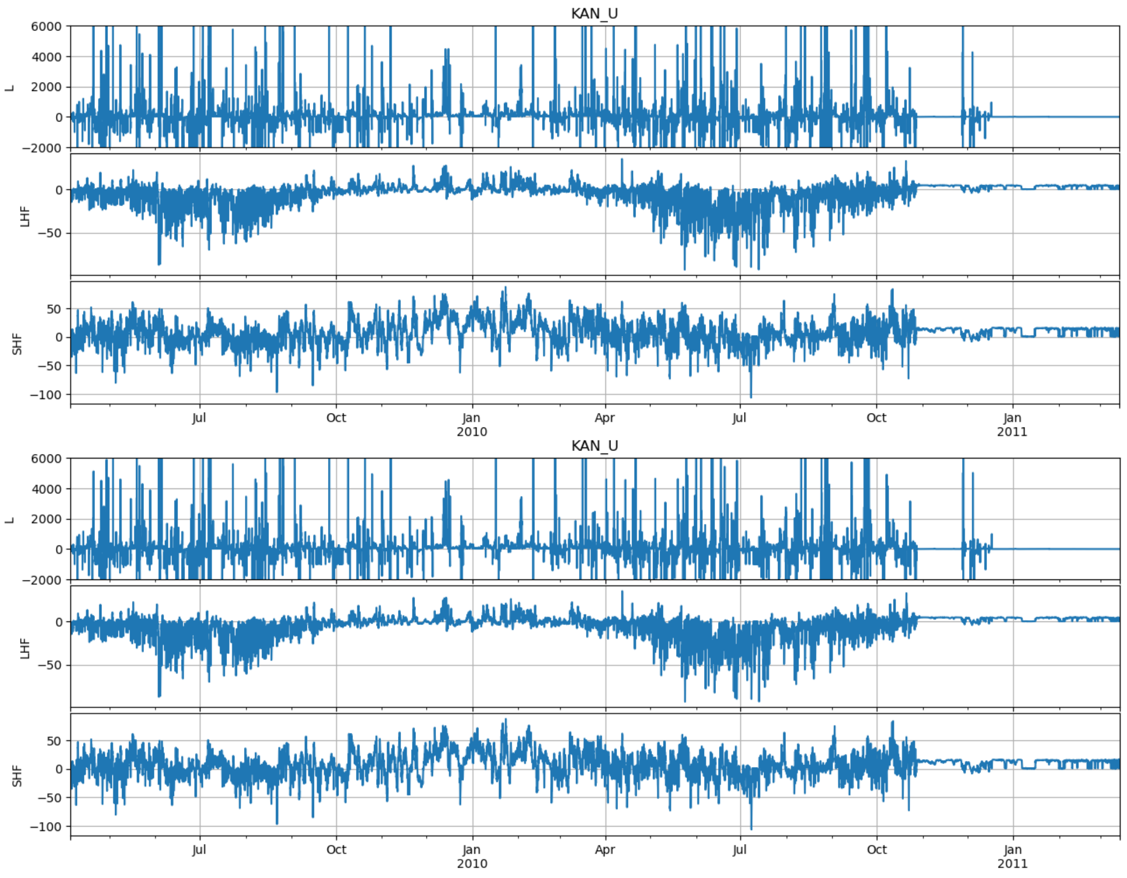


Figure 8. Above: Summary of `L`*,* `LHF` *&* `SHF` *output from optimized code, 200 layers. Below: Summary of* `L`*,* `LHF` *&* `SHF` *output from original code, 200 layers.*

Beyond the random big differences in `L`, there existed one more random error in the code, earlier mentioned in Release 1. This caused the program to warn about overflows and division with zero. It was also the reason for not being able to implement Numba in one code part as this led to the program crashing. The reason for these values to occur could be because some data values or computations caused zero values, which often seemed to result in the `Tsurf` variable being assigned 0. It is believed that the overflow and zero division are connected. An overflow in one variable could likely cause another computation, containing that very large value, to result in a very small value, which in another operation is divided with, causing the zero division. It could also be worth investigating whether the static values assigned to some variables when a certain condition is not met (described under 5.2.1 Release 1: Specification) are set correctly or if any of these could cause the overflow. Furthermore, another issue was found, namely that the snow thickness was always larger than zero, meaning that `RoughSurf` was never called, even though the snow thickness should be able to be zero.

To further study the results, a comparison was made between the outputs from the original Python code, the optimized Python code, and the original MATLAB code. Some of the results are presented in Figure 9 and 10. Worth noting, the time axis in the figures differ slightly. It was seen that the two Python code outputs were similar to each other but differed from the original MATLAB code. Mainly, gaps in the Python code outputs existed for some variables, for example in the temperature graphs presented in Figure 9. However, where there were no gaps, one can see that the temperature values were consistent between the MATLAB and Python results. Not all variables had gaps in the plots. One example is the computed latent heat fluxes seen in Figure 10. Here, a greater difference in the computed values between the MATLAB simulation and Python simulations is instead visible. If looking at the minimums and maximums in the data, the two Python simulations resulted in similar values (minimums -170.5 and -169.6, maximums 50.6 and 51.3), whereas MATLAB led to greater values (minimum -194.6, maximum 61.5). However, the overall pattern of the three graphs was the same.

In summary, one can say that the original and optimized Python code led to very similar results but differed from the original MATLAB code. Considering this, it has been further validated that the optimization performed in this study so far has not affected the Python code functionality. A takeaway from this comparison is that an issue must have occurred when the code was migrated from MATLAB to Python. If GEUS use the Python models for simulating the Greenland ice sheets, the models should first be revised from a functionality perspective. Making sure the computations yield correct results and that no data input values lead to gaps in the plots is of importance.

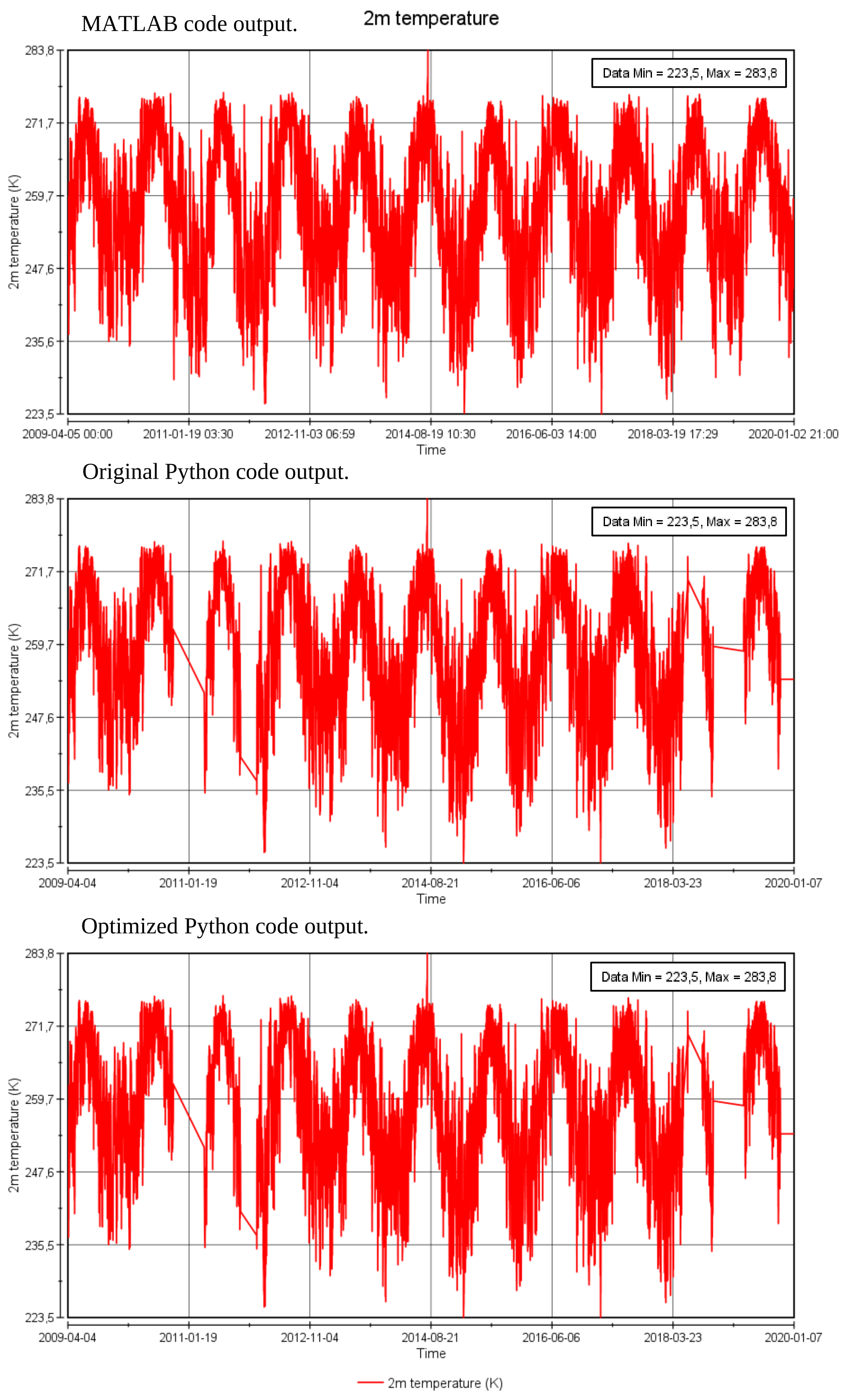


Figure 9. Temperature outputs. Above: MATLAB output. Middle: Original Python output. Below: Optimized Python output.

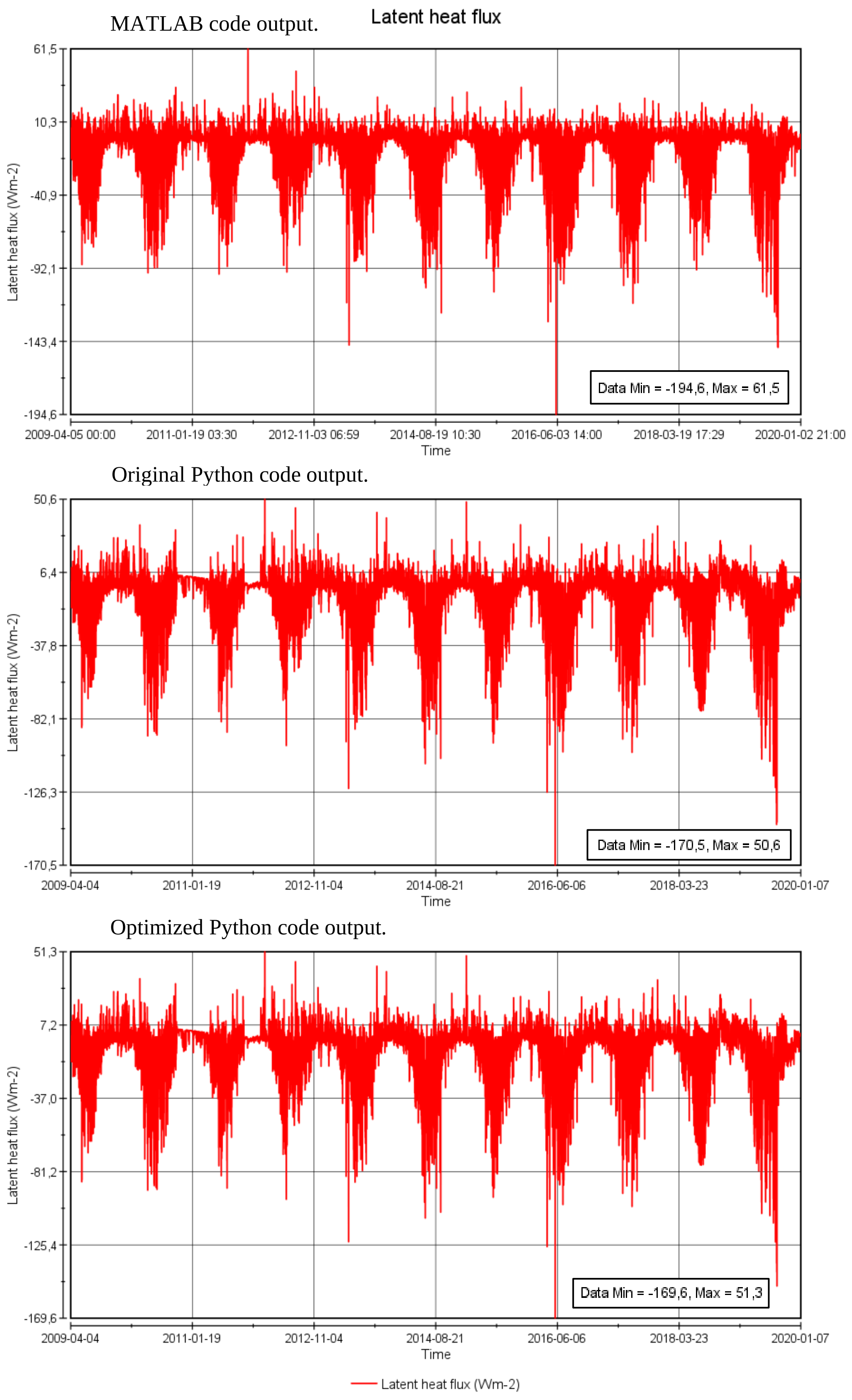


Figure 10. Latent heat flux outputs. Above: MATLAB output. Middle: Original Python output. Below: Optimized Python output.

5.6 Release 4: Performance Optimizing the Firn Evolution model

For Release 4, the focus was moved to the Firn Evolution model. The difference between the Firn Evolution model and the SEB model is discussed under 2.2 The Models of this Study. In summary, the difference is that SEB numerically computes the surface conditions, whereas Firn Evolution uses data calculated beforehand. During this project, preprocessed data containing this information was not available. Instead, the parameter values were extracted from SEB, namely the values of sublimation, meltwater, snowfall, and surface temperature for each time stamp. The data used was from the KAN_U and KAN_M weather stations, when simulating for 200 layers. In the earlier releases, it had been noted that differences were easier to detect when simulating for 200 layers instead of 50. Even though simulating for 200 layers takes more time, it was decided to mainly perform 200 layered simulations in the following release.

5.6.1 Release 4: Specification

The Firn Evolution model is built similarly to the SEB model, starting with initializing constants and variables, and loading weather input data (this is where information extracted from SEB model runs were used as input). The function `subsurface` is then called directly. Each time step is simulated with computed information about snow, ice, and water content, grain size, surface temperature and more about the layers. The main function of *main_firn.py*, the `run_GEUS_model`, is called for each weather site in a list. This makes it possible to get the surface information for several weather stations through only one call.

To understand where the bottlenecks of Firn Evolution exist, a profiling was performed. As seen in Table 12 and 13, the `update_tempdiff_params` function consumes most time. All functions in the two tables are part of the *lib_subsurface.py* script. From the profiling one can see that compared to the SEB model, the Firn Evolution model runs much faster. This was expected since fewer steps are performed and since the most time-consuming parts of the SEB model are not called from Firn Evolution. When running all data rows for both weather stations KAN_U and KAN_M as presented in Table 13, more time is needed, and the bottlenecks are more obvious.

Table 12. Result from initial profiling of Firn Evolution code. 6000 rows, KAN_U dataset.

Number of calls	Total time (in sec)	Per call (tot.) (in sec)	Cumulative time (in sec)	Function
6000	3,037	0,001	4,358	`update_tempdiff_params`
6000	2,562	0,00	2,565	`merge_small_layers`
6000	0,823	0	0,823	`calc_snowdepth1D`
6000	0,681	0,00	1,021	`superimposedice`

Table 13. Result from initial profiling of Firn Evolution code. 94323 rows KAN_U, 29270 rows KAN_M.

Number of calls	Total time (in sec)	Per call (tot.) (in sec)	Cumulative time (in sec)	Function
123 593	68,03	0,001	89,71	`update_tempdiff_params`
123 593	56,15	0,00	56,15	`merge_small_layers`
123 593	17,70	0	17,70	`calc_snowdepth1D`
123 593	14,84	0,00	14,84	`superimposedice`

The goal of this release is to investigate if `update_tempdiff_params` can be sped-up while maintaining its purpose of updating the thermal capacity and conductivity of the subsurface as well as calculating the subsurface heat flux calorific capacity [38] (for the purpose of this thesis, it is not necessary to understand the meaning of these variables). As the Firn Evolution model will be called for each weather site in a site list, it will be investigated if this procedure can be made faster through parallelization.

5.6.2 Release 4: Implementation

Initially, some rearrangement had to be performed to get the simulation running, such as reading the extracted data of melt, sublimation, snowfall, and surface temperature from SEB. Once the model was running, an attempt to parallelize was initiated. The `parallel` method from Joblib was imported and implemented in the loop calling the Firn Evolution model for each site in the list. Simulating more than one weather site is of interest to get an overview of Greenland. If several sites can be run in parallel, the simulation can be more extensive while the execution time is reduced. Even though the number of cores on the laptop was limited, an eventual speed up from parallelizing demonstrates that it could be of interest to run the simulation on multiple cores. The parallelization was done as follow. Instead of calling the simulation `run_GEUS_model` for each site in a list through a for-loop (Code block 10), the simulation calls were done for each site in parallel using `Parallel` (Code block 11). The filename, which was hard coded in the original version, was removed for easier maintenance.

```
def run_main_firn_old(site_list):
    run_name_list = []
    for site in site_list:
        filename = "Firn viscosity/Input files/" + site + ".csv"
        run_name = run_GEUS_model_old(site, filename)
        run_name_list.append(run_name)
    return run_name_list
```

Code block 10. Original way of calling `run_GEUS_model`, for each site in a for-loop.

```
def run_main_firn_parallel(site_list):
    num_cores = multiprocessing.cpu_count()
    run_name_list = Parallel(n_jobs=num_cores, verbose=10)(delayed(run_GE
US_model_opt)(site) for site in site_list)

    return(run_name_list)
```

Code block 11. Optimized way of calling `run_GEUS_model`, for each site in parallel.

Furthermore, the most time-consuming part of the Firn Evolution model was investigated, `update_tempdiff_params`. The function was very long and could not be compiled through Numba. Instead, the most time-consuming part was moved to a `@jit`-decorated function. This part included computations in a for-loop consisting of several variables. Compiling the part in Numba was possible and enabled speeding up a function that otherwise could not be `@jit`-decorated. The reason for not being able to compile some functions in Numba is due to the use of functions not defined in Numba or non-supported language features that Numba cannot understand [63].

5.6.3 Release 4: Validation

Calling the model in parallel resulted in a big speed up. To test simulating different number of sites, three lists were created using KAN_M and KAN_U data one or multiple times. This made it possible simulating for two, four respectively eight weather sites. The execution times are seen in Table 14, where the parallel calls are seen to run faster than the serial ones. With two sites in the list, the parallelization showed 1.5 times speed up, with four sites 2.0 times speed up, and for eight sites the effect was even bigger with 2.6 times speed up saving 57.8 seconds on average. However, the parallel simulation times varied more greatly with larger standard deviations. Moreover, simulating in parallel affected the computer greatly as the fan was running on a very high speed and made noises. If having a computer with larger capacity, parallelizing for more stations is believed to be possible and reduce the execution time a lot.

Table 14. Execution CPU time, original versus parallelized code (mean ± standard deviation). 6000 rows, KAN_U and KAN_M.

Number of weather sites	Version of code	Total time (in sec)
2	Original	23,53 ± 0,62
2	Parallelized	16,03 ± 6,1
4	Original	51,71 ± 1,0
4	Parallelized	25,53 ± 6,9
8	Original	93,77 ± 0,66
8	Parallelized	36,0 ± 6,7

The effect from implementing Numba had a great effect also in this release. If starting with comparing runs in series, the implementation of Numba for two sites led to 1.3 times speed up, for four sites 1.3 times speed up, and for eight sites 1.2 times speed up. If comparing runs in parallel, the Numba implementation led to 1.1 times speed up for two sites, 1.2 times speed up for four sites, and 1.3 times speed up for eight sites. The times can be seen in Table 15. If comparing executing the original code in serial, the second column, with running the Numba optimized code in parallel, the fifth column, the differences were big. For two and four sites it led to 1.4 respectively 2.5 times faster executions. For eight sites, the largest speed up so far was seen. The parallelized Numba implemented code ran 3.2 times faster than the unoptimized run in series. Thus, performance optimizing the Firn Evolution model, by running it in parallel and compiling parts in Numba, had great effect. Due to limited time resources, no further Numba implementations were done. As parts of `update_tempdiff_params` and `merge_small_layers` still consumed much time, it is believed that a further speed up could be reached if compiling more code through Numba.

Table 15. Execution CPU time (mean ± standard deviation), Firn Evolution model. Executed in serial/parallel, with/without Numba. 6000 rows, KAN_U and KAN_M.

Number of weather sites	Original code, serial (in sec)	Numba in code, serial (in sec)	Original code, parallel (in sec)	Numba in code, parallel (in sec)
2	$26,3 \pm 0,8$	$20,8 \pm 0,6$	$21,9 \pm 6,0$	$19,3 \pm 6,3$
4	$52,9 \pm 0,7$	$40,6 \pm 1,1$	$25,1 \pm 6,5$	$21,4 \pm 6,6$
8	$101,3 \pm 0,7$	$81,6 \pm 0,5$	$42,8 \pm 6,2$	$31,8 \pm 7,2$

5.7 Release 5: Adapting the SEB model to the CARRA data

The aim of this study has been to evaluate performance optimization strategies to enable upscaling a simulation model with more data. The second part of upscaling with CARRA data will now be initiated. For this purpose, the SEB model has been used.

5.7.1 Release 5: Specification

The first task was to extract the data that was stored remotely in GEUS' files. Once extracted, the data had to be preprocessed to match the scripts of the simulation models. In Table 16, the variables needed for the simulations are presented together with whether they exist in the CARRA data or not, if they must be derived from another variable, as well as if a unit conversion was necessary.

Table 16. CARRA data overview.

Variable	Exists in data	Derived from	Unit conversion
Air pressure	Yes	-	Pa to hPa
Albedo	Yes	-	% to 0-1 range
Height, temperature	No	Always 2 m.	-
Height, wind speed	No	Always 10 m.	-
Height, humidity	No	Always 2 m.	-
Longwave Radiation Downwards	Yes	-	Joule/m^2 to Watt/m^2
Longwave Radiation Upwards	No	Longwave Radiation Downwards, Temperature	-
Rainfall	No	Total precipitation, Temperature	Kg/m^2 to m w. eq.
Shortwave Radiation Downwards	Yes	-	Joule/m^2 to Watt/m^2
Shortwave Radiation Upwards	No	Albedo, Shortwave Radiation Downwards	-
Snowfall	No	Total precipitation, Temperature	Kg/m^2 to m w. eq.
Temperature	Yes	-	K to °C
Total precipitation	Yes	-	12H time step to 3H time step

5.7.2 Release 5: Implementation

To get the simulation running with CARRA data, some preprocessing had to be done. An extraction script was created in Python, in which the reading from CARRA files was done together with deriving and transforming variables. To enable a comparison between simulating AWS respectively CARRA data, the same time span had to be used. There was only a small amount of data from KAN_U available at the time, where all variables existed for the same dates in both the AWS and CARRA set. No data existed for the same dates for KAN_M. Because of this, only KAN_U data was studied.

Initially, missing data and null values were looked for. For longwave radiation downwards, no data existed after the 3rd of October 2010, whereas data for the other variables existed until the 30th of June 2022. This limited the analysis to only looking at the data before October 2010. Between some dates, from the 24th to the 30th of June 2000 and the whole month of December 1992, no values existed for the shortwave radiation downwards. These null values were substituted with zeros, but as these were not part of the time interval for the comparison with the AWS data, it did not impact the result discussed later. Missing values also existed in total precipitation due to different lengths of time steps. To be able to derive rainfall and snowfall from total precipitation, the precipitation variable itself had to be processed as it had a time step of 12 hours and all other data existed for each third hour. For each time step, t, the CARRA data contained total precipitation data from hour t+18 and t+6. By subtracting the value at t+6 from t+18, the accumulated precipitation at hour t+12 was given, thus giving the 12-hourly time step. Thereafter, to get data for every third hour, data imputation was performed. The missing values were filled with an estimation of the precipitation every third hour. A linear approximation was performed, equally dividing the total precipitation at hour 12 to each third hour within that time step. To further know whether the precipitation was snowfall or rainfall, the temperature for that time step was investigated. If above 0 °C the precipitation was interpreted rain, otherwise snow. Lastly, the snowfall and rainfall had to be converted from the unit kg/m^2 to m w. eq., where 1000 kg/m^2 equals 1 m w. eq.

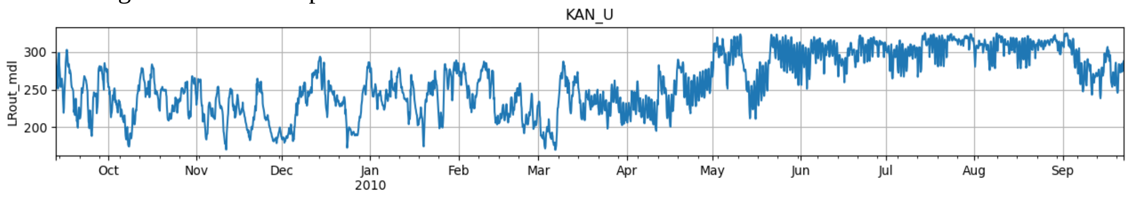
The heights at which the collection of wind speed, temperature and humidity was taken had to be set as well. These variables are the same for all the CARRA data and were set to the standard heights of 2 meters respectively 10 meters for wind speed. Furthermore, the longwave radiation upwards had to be derived from temperature and longwave radiation downwards. The longwave radiation downwards had to be converted from Joule/m^2 to Watt/m^2 and temperature be kept in Kelvin when doing the computation. The shortwave radiation upwards was derived from multiplying the shortwave radiation downwards with the albedo [11]. Beyond these transformations, some scaling and unit conversions were made to match the simulation model.

5.7.3 Release 5: Validation

One reason for optimizing the simulations was to enable scaling up with the CARRA dataset. The result from that upscaling will now be presented, starting with a comparison of the CARRA and AWS output, followed by the effect from optimization the SEB model. Comparing outputs from the two different data sources was of interest as model data (such as CARRA) can differ from purely observed data (like AWS). By observing how the CARRA data affects the simulation one can see if it is possible to use the data, or if smaller differences in one variable exist that lead to bigger differences in the outcome (Baptiste Vandecrux, April 2023, oral communication).

First and foremost, the preprocessing and adaptation to the SEB model was claimed successful as the model could simulate the CARRA data. When comparing the CARRA and AWS outputs, some differences and similarities existed. Generally, the energy input seemed to be less in the CARRA dataset than in AWS. If comparing one of the data inputs, the longwave radiation upwards in Figure 11, the CARRA and AWS data seemed to match rather good for the period. Overall, the graphs follow the same pattern and reaches similar values at the same times.

CARRA data.
Longwave radiation upwards.



AWS data.
Longwave radiation upwards.

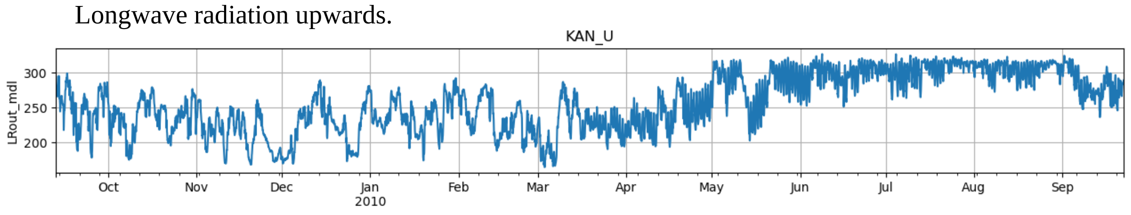


Figure 11. The input data variable longwave radiation upwards plotted for CARRA (above) and AWS (below), during the time September 2009 to October 2010.

The shortwave radiation downwards on the other hand showed larger differences between the two datasets. The overall pattern was the same, see Figure 12, but the CARRA data reached slightly higher in its maximums, up to around 50 W m^{-2} higher, while at the same time not giving as high values as the AWS data for some dates. If for example looking at August and September 2010, CARRA has greater dips in the shortwave radiation downwards. A difference was also seen in the computed meltwater. The AWS data produced more meltwater than CARRA, still, the patterns were the same and melt occurred at the same time for both datasets, just to different extents.

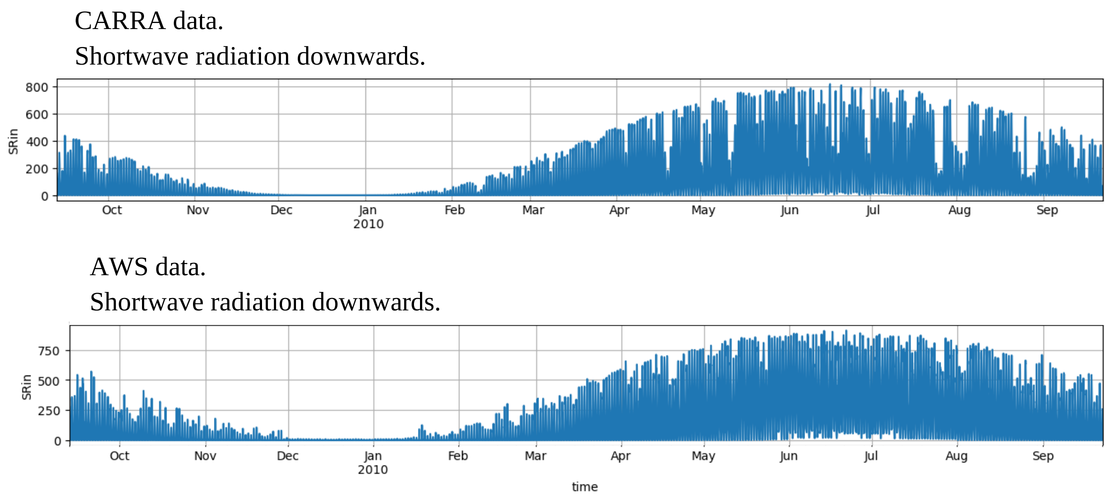


Figure 12. The input data variable shortwave radiation downwards plotted for CARRA (above) and AWS (below), during the time September 2009 to October 2010.

When studying the outputs, for example the surface temperature presented in Figure 13, a big difference was seen as well. The deeper layers were more similar, whereas in the top surface one could see that the AWS data showed warmer temperatures, and a bigger change in temperatures in July, August, and September 2010. There was also a difference on the top surface where the AWS data resulted in bigger variations in height and where the melt was bigger. This is also visible in Figure 14, showing the liquid water content in the layers. The AWS data resulted in a higher water content towards the end of the period than the CARRA data did. However, the CARRA result showed the same overall pattern of temperature changes and increased liquid water content towards the end of the period, just not to the same extent.

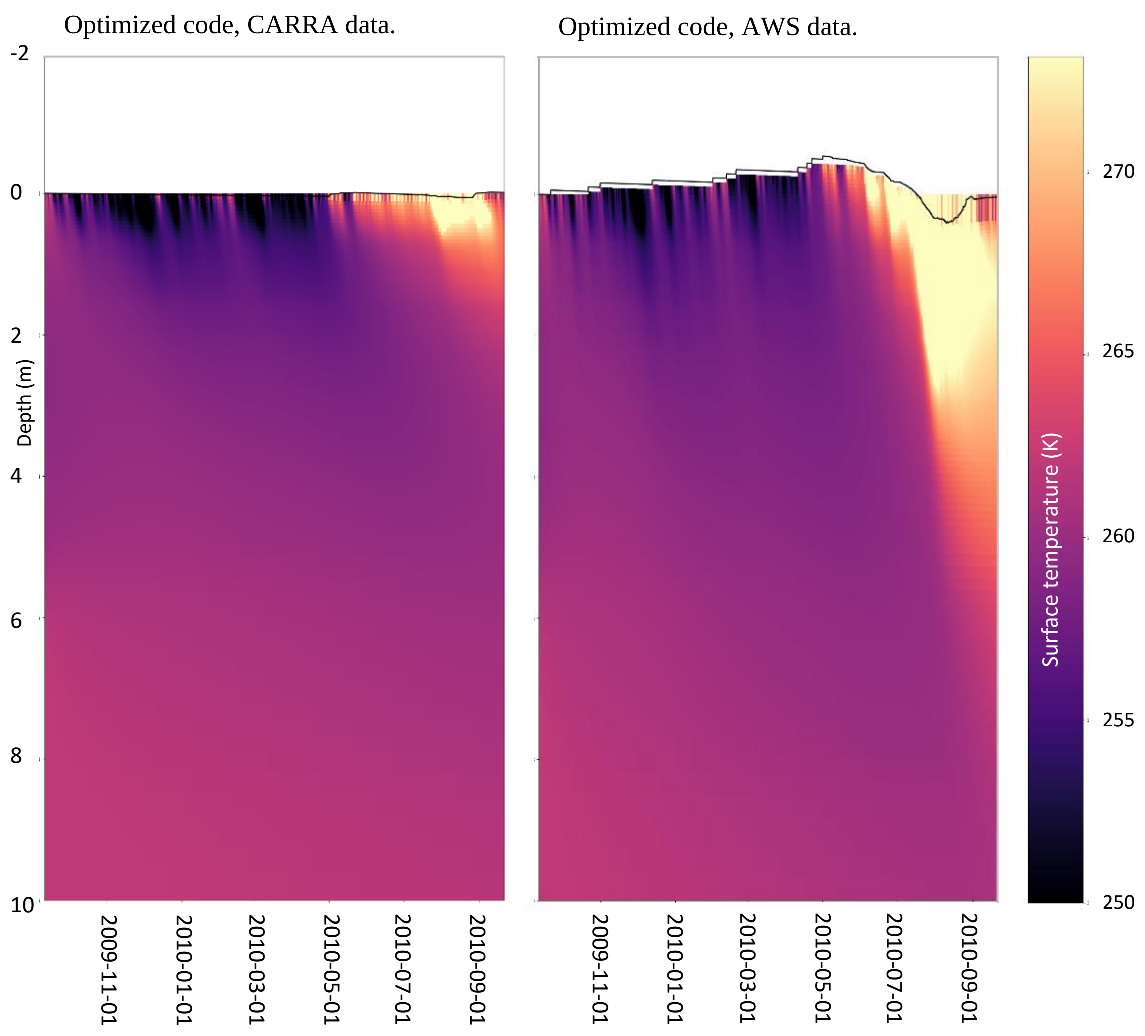


Figure 13. The surface temperature (K) plotted for KAN_U using optimized code. To the left is when using CARRA data, to the right AWS data.

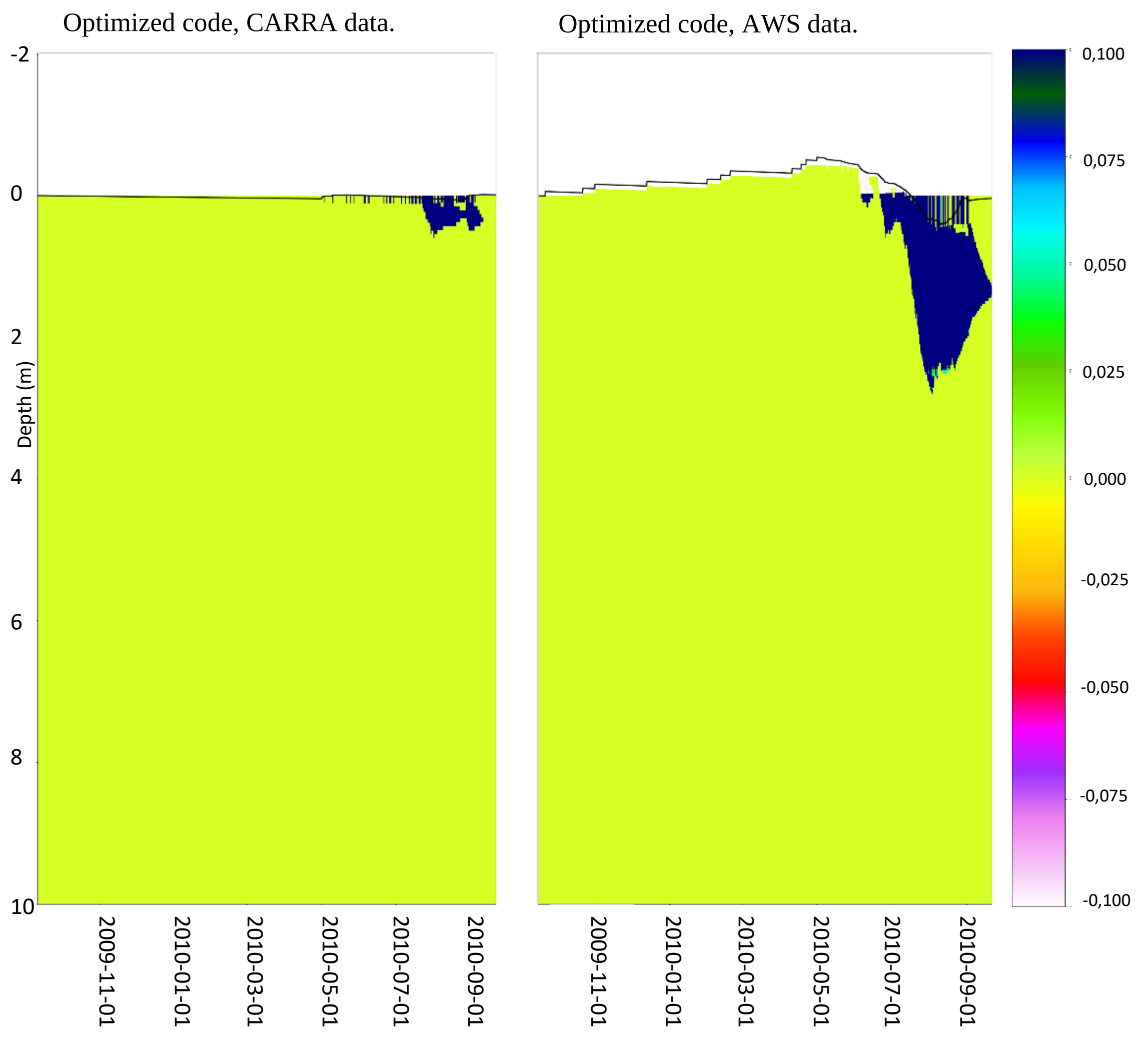


Figure 14. The liquid water content (mm m^{-3}) plotted for KAN_U using optimized code. To the left is when using CARRA data, to the right AWS data.

Finally, the execution time when simulating the ice sheet with CARRA data from KAN_U was measured for the unoptimized and optimized version of SEB, see Table 17. When running the simulation with 200 layers, for 3000 data rows, the optimized code ran 1.5 times faster on average. For 6000 data rows, the execution was 1.7 times faster. When simulating with AWS data, the corresponding speed up was 1.9.

Table 17. Execution CPU time, original versus optimized code (mean ± standard deviation). CARRA, KAN_U dataset.

Number of rows	Original code (in sec)	Optimized code (in sec)
3000	$71,4 \pm 0,4$	$46,7 \pm 1,2$
6000	$125,5 \pm 12,7$	$72,1 \pm 4,4$

6. Discussion

As the spiral model involved a validation phase, some discussions have already been made while presenting the results. This section instead aims to provide a more summarizing discussion.

6.1 The Pareto Principle and Spiral model

Throughout the project, the Pareto Principle was repeatedly reflected on. It was indeed smaller parts of the codes that consumed most time and by developing these parts, a big change in time was visible. When working with the SEB model, the by far biggest speed up was seen after the first release. After the first release the execution ran 1.6 times faster, after the second release 1.2 times faster compared to the Release 1 code, and after Release 3, 1.1 times faster compared to the Release 2 code. The effect from optimizing was thus biggest in the beginning and then decreased. By identifying the most time-consuming part and putting effort into developing that section, the most effect could be received. If Numba compiled functions had been continued to be implemented in yet another release, the effect might have been even less, even though the same amount of effort would have been put into it. The decreased effect from Release 3 was therefore a good indicator to move on with the project. Similarly for the Firn Evolution model, by putting little effort into running the simulation in parallel without changing much of the code, a big speed up was seen.

It is believed that Sommerville's [1] spiral model made it easier to identify the Pareto Principle in the study, as the validations of each release enabled an easy comparison of the results. Beyond this, the spiral model offered a structured approach for working with the code. Breaking down the code into smaller sections made it possible to get a good understanding without getting overwhelmed. However, as the specification and operation phases of this project were very similar in each release, these parts sometimes felt redundant. At the same time, they initiated time for reflection and, especially the specification, created a solid ground for understanding the code. The spiral model thus felt suitable for the project.

6.2 Performance Optimization through Numba and Parallelization

In the beginning of the study, two types of implementations aiming to speed up the code were identified. The first was to implement Python packages, the second to use proper code hygiene, data structures and algorithms. In the first releases the first type, to implement Numba, was seen to have the biggest impact on speeding up. As the purpose of Numba is to speed up Python code by compiling it in low-level language, this was an expected but successful result. The Numba compilation had great effect when compiling code sections including many, yet simple, computations of addition, multiplication, and division. Compiling large functions in `@jit`-decorated functions was not possible, as dependent variables, unsupported language, and functions unknown to Numba existed. Instead, a better result was seen if moving smaller, time-consuming code parts from bigger functions to `@jit`-decorated ones. Also, this did not lead to faulty results which were hard to trace back. However, cases existed where Numba did not have desired effect. It is thus important to reflect on where Numba, and similarly NumPy, is being implemented to be sure it is appropriate. The reason for the Numba implementations to not result in larger speed ups than roughly twice the speed is likely due to the complexity of the models and their many iterations. Numba enables compiling code sections in machine code, but it might be necessary to rewrite the algorithms more thoroughly which likely demands good glaciology knowledge. Furthermore, the Numba implementations were mainly done in the SEB model and only in a small part of the Firn Evolution model. It is believed that by moving more code sections from the Firn Evolution model to `@jit`-decorated functions, a bigger speed up could be gained. If linking back to the Pareto Principle, maybe only little more effort is required to give a good result, if continuing to look at the most time-consuming parts. Or, on the contrary, the largest speed up might already have been given, meaning that even if more effort is put, no big effect will be seen.

In the fourth release, parallelizing the calls to the Firn Evolution model showed a great impact on reducing the execution time. This can be considered being of the second implementation type, to change the code structure. If studying the parallelization from the view of Amdahl's law, the estimated performance would be four times the speed, as this is the laptop's number of cores. As the whole simulation for each weather site could be run in parallel, with no part having to be performed in sequence, all four cores could be used instead of just one, thus theoretically leading to a four times faster execution. However, as brought to light by the Roofline model, there is also a limitation due to the memory bandwidth, as this parallelization is of the shared memory type. This is likely why the actual speed up from parallelizing ranged between 1.5 and 2.6 times the execution time, and when combined with Numba could reach 3.2 times speed up depending on the number of weather sites. Accessing the memory and transporting data also needed processing capacity, making it impossible to only use the processors for computations. Furthermore, the biggest increase in performance was seen when parallelizing the simulation for eight weather sites using eight threads. Without Numba implementations, the parallelized execution ran 2.6 times the serial execution time, and with Numba, 3.2 times faster.

One could believe that the biggest increase would have been when simulating four sites, as each site could be run on one core. The reason for receiving such a good speed up when simulating eight sites could be due to a good overlap between computations and memory accesses emerged. As infinite amounts of data cannot be loaded at the same time, it is likely to believe that while loading some data, computations could be done in a good balance. Considering the great outcome of running the Firn Evolution model in parallel, and especially when combined with Numba implementations, it is likely that the SEB model can also benefit from parallelizing simulations for different weather sites. The reason for why parallelization was not implemented to SEB in this study, is due to time constraints.

Furthermore, the larger standard deviation in parallel times showed that a greater variation in execution times existed and that running in series led to more predictable, yet longer, execution times. However, it can be considered worth having a larger deviation in the execution times, if it means getting faster simulations.

The attempts to speed up by changing data types, reducing the imports to the script, or rearranging code did not have big impacts, some even led to slower executions. Reducing the repetition of code sections had little effect on the time but led to a more maintainable and understandable code. The sections of the code that were numerically iterated through multiple times were not possible to comprehend or rewrite more efficiently, other than making some computations Numba compiled. The for-loops were too big to comprehend, and for rewriting them more glaciological knowledge would be needed. Furthermore, examining different data types and packages such as pandas and NumPy had already been implemented in the code to enable an efficient handling of arrays and data. Optimizing the code in this sense had thus already been done and the result from it cannot be accounted for.

Overall, the performance optimization might enable running larger simulations for bigger areas simultaneously, if using computers with larger capacity. A better overview of the Greenland ice sheet could through this be received in less time. Simulating all CARRA data would still be time-consuming (and create outputs different from AWS), but if running on multiple processors the execution time could be reduced greatly. Beyond enabling for simulating more data, the faster execution time might also make it possible to increase the details and granularity of the models. With smaller grid spacings the accuracy of the model outcome can possibly be higher, which with an efficient model might not lead to largely increased execution times.

6.3 Simulating for Different Data

Initially, it was believed that different parts of the SEB model might be the most time-consuming depending on the dataset being used. When comparing simulations with KAN_U respectively KAN_M data this was not the case. Even though the weather conditions at the sites differed, the same code sections were seen to be the bottlenecks. Worth mentioning, this could be because of the bug in the code, leading to the snow thickness always being bigger than zero.

In Release 5, differences between the AWS and CARRA datasets were found even though it was possible to see the same overall changes in the ice sheet. It is hard to say what caused these differences, but since CARRA is modelled data created by observations and model computations, the reason might be due to simplifications done when creating the data. The estimation made when computing the 3-hourly precipitation data might have affected the result as well, but it is not believed being the reason for the larger differences as the same accumulated precipitation should be similar. Also, since the comparison was done with only a small portion of data it might not be representative for all data.

Considering the result, one might be able to use the CARRA data to get an idea of what has happened in the surface, but not on a detailed level. CARRA could be used to see overall patterns, but other data might be necessary if more exact outputs are needed. Even though the data comparison might not have resulted in the wished output, a good speed up was seen when simulating the CARRA data with the optimized SEB model. The speed up was not as big as for the AWS data which had a 1.9 times faster execution compared to CARRA’s 1.7 times faster (for 200 layers and 6000 data rows), but the performance had increased. The smaller speed up could be due to how the CARRA data is loaded. A faster way could be to only load the rows asked for instead of loading all rows and then selecting a smaller portion of it, as CARRA contains a lot of data, but this could be investigated further.

7. Conclusions

The purpose of this study was to investigate how a simulation model can be performance optimized in terms of reducing execution time and how this can enable scaling up with more data. As the simulation consisted of many numerical iterations to compute the physics of the ice sheets, no suitable way to reduce or remove these loops were found. Moreover, since efficient packages for handling data arrays were already implemented in the code, this study could not examine the effects these might have had either, even though they are believed to have been positive. Other implementations, such as changing the imports, did not make the execution more efficient. Instead, the by far most efficient strategy was to move selected code sections to Numba compiled functions and to execute the simulations of several weather sites in parallel. When simulating CARRA, the Numba implementations led to 1.5 respectively 1.7 times faster executions. For the AWS data, the Numba implementations led to between 1.9 and 2.1 times the speed, and the parallelization between 1.5 and 2.6 times the speed, depending on the number of layers, data rows and weather sites. The greatest speed up was given when combining the two approaches. When simulating AWS data with the Numba optimized code for eight weather sites in parallel a 3.2 times faster execution was seen. Thus, if combining Numba with simulating several weather stations in parallel, the biggest increase in performance can be given. As Numba was mainly implemented in the SEB model and parallelization only in the Firn Evolution model, it is believed that both models could benefit from more implementations of the other kind.

When it comes to scaling up the simulations with more data, parallelization is believed to have most effect. If running on multiple processors, the parallelization can create a possibility to simulate larger parts of the Greenland ice sheets simultaneously. The optimization through Numba will also contribute to this, but when scaling up further it is believed that the parallelization could stand for the biggest effect. Simulating the whole Greenland ice sheet with Numba implemented and parallelized code on a laptop like mine will most likely not be possible, but on a computer or server with more processors, it might be. Further, as the outputs from simulating AWS and CARRA data differed, the CARRA dataset might not be the one to use for an upscaling. Instead, CARRA could be used for creating overviews of occurring changes when only a perception of what has happened is necessary. Though, due to the differences, CARRA should not be used when accurate and detailed results are needed.

Through this study it has been shown that it is possible to performance optimize two ice sheet simulation models using a laptop with only four cores. Based on the findings, the best effect is received when combining implementations of Numba and executing the simulations in parallel for several weather sites. Considering the difference between the CARRA data and AWS data outputs, it might not be desired to simulate all CARRA data. But, if one wishes to, running the optimized simulation on multiple processors with higher capacity could be possible to simulate larger parts of the Greenland ice sheet. An overview of patterns and trends in the glacier's surface since 1991 could thus be given, hopefully contributing to the climate change research.

7.1 Future studies

The errors and issues found through the study are important to address in the future if the two ice sheet models are to be used. Firstly, the difference between the original MATLAB code and the Python codes must be resolved. Secondly, making sure that the code runs smoothly even when missing data exist. And thirdly, the errors discussed under 5.5.2 Problems and Issues Encountered should be handled. These could be suitable for future studies or projects.

Moreover, since the pandas and NumPy libraries had already been implemented in the code before this study, it was not possible to examine the contribution these may have on optimizing code. Therefore, a study that investigates the effect from implementing these libraries in simulation models could be conducted.

References

[1] Sommerville, I. (2011), Software engineering. 9th. ed., International ed., Addison-Wesley: Harlow.

[2] Cardoso, JMP., Coutinho, JGF., Diniz, PC. (2017), "High-performance embedded computing" in: Embedded Computing for High Performance. Elsevier: Place of publication not identified, pp. 17–56.

[3] Källén, M. (2021), Towards Higher Code Quality in Scientific Computing. Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2000. ISBN 978-91-513-1107-4, Acta Universitatis Upsaliensis: Uppsala.

[4] Lam, SK., Pitrou, A., Seibert, S. (2015), "Numba: a LLVM-based Python JIT compiler" in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. Association for Computing Machinery: New York, pp. 1–6.

[5] Fowler, M. (1999), Refactoring: improving the design of existing code. Addison-Wesley: Reading, MA.

[6] Boehm, B., Basili, VR. (2001), "Software defect reduction top 10 list", Computer, vol. 34, no. 1, pp. 135-137.

[7] McKinney, W. (2012), Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. O'Reilly Media, Inc: Beijing.

[8] Bueler, E. (2022), "Performance analysis of high-resolution ice-sheet simulations", Journal of Glaciology, 2022-12-14, pp. 1–6.

[9] Tillenius, M., Larsson, E., Lehto, E., et al. (2015), "A scalable RBF–FD method for atmospheric flow", Journal of computational physics, vol. 298, pp. 406–422.

[10] GEUS (n.d.), The Greenland ice sheet. Available online: https://eng.geus.dk/nature-and-climate/the-greenland-ice-sheet (accessed 25 January 2023).

[11] Van As, D., Van Den Broeke, M., Reijmer, C., et al. (2005), "The Summer Surface Energy Balance of the High Antarctic Plateau", Boundary-layer meteorology, vol. 115, no. 2, pp. 289–317.

[12] Langen, PL., Fausto, RS., Vandecrux, B., et al. (2017), "Liquid Water Flow and Retention on the Greenland Ice Sheet in the Regional Climate Model HIRHAM5: Local and Large-Scale Impacts", Frontiers in earth science, vol. 4, article 110.

[13] Ahlkrona, J. (2016), Computational Ice Sheet Dynamics: Error control and efficiency. Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1368. ISBN 978-91-554-9562-6, Acta Universitatis Upsaliensis: Uppsala.

[14] Van den Broeke, MR., Enderlin, EM., Howat, IM., et al. (2016), "On the recent contribution of the Greenland ice sheet to sea level change", The Cryosphere, vol. 10, no. 5, pp. 1933–1946.

[15] Alley, RB., Clark, PU., Huybrechts, P., et al. (2005), "Ice-Sheet and Sea-Level Changes", Science, vol. 310, no. 5747, pp. 456–460.

[16] ASOC (n.d.), Antarctic Ice and Rising Sea Levels - Antarctic and Southern Ocean Coalition. Available online: https://www.asoc.org/learn/antarctic-ice-and-rising-sea-levels/ (accessed 13 April 2023).

[17] Hörhold, M., Münch, T., Weißbach, S., et al. (2023), "Modern temperatures in central–north Greenland warmest in past millennium", Nature, vol. 613, no. 7944, pp. 503–507.

[18] Stephens, GL., O'Brien, D., Webster, PJ., et al. (2015), "The albedo of Earth", Reviews of Geophysics, vol. 53, no. 1, pp. 141–163.

[19] Buizert, C. (2013), "ICE CORE METHODS | Studies of Firn Air" in: Elias, SA., Mock, CJ. (eds), Encyclopedia of Quaternary Science (Second Edition). Elsevier: Amsterdam, pp. 361–372.

[20] Cuffey, KM., Paterson, WSB. (2010), The Physics of Glaciers. Butterworth-Heinemann: Amsterdam.

[21] Harper, L. (2018), Glaciers, Ice Sheets, and More: A Primer on the Different Types of Polar Ice. Available online: https://news.climate.columbia.edu/2018/02/05/glaciers-ice-sheets-polar-ice/ (accessed 13 February 2023).

[22] Thirumaleshwar, M. (2006), "Chapter 1. Introduction and Basic Concepts" in: Fundamentals of Heat and Mass Transfer, Pearson: Place of publication not identified.

[23] American Meteorological Society (2016), Monin-obukhov similarity theory. Available online: https://glossary.ametsoc.org/wiki/Monin-obukhov_similarity_theory (accessed 20 February 2023).

[24] American Meteorological Society (2016), Obukhov length. Available online: https://glossary.ametsoc.org/wiki/Obukhov_length (accessed 20 February 2023).

[25] Galindo, A. (2023), What is Radiation?. Available online: https://www.iaea.org/newscenter/news/what-is-radiation (accessed 14 February 2023).

[26] American Meteorological Society (2012), Roughness length. Available online: https://glossary.ametsoc.org/wiki/Roughness_length (accessed 12 May 2023).

[27] Paterson, WSB. (1994), "9 - Structures and Fabrics in Glaciers and Ice Sheets" in: Paterson, WSB. (eds), The Physics of Glaciers (Third Edition). Pergamon: Amsterdam, pp. 173–203.

[28] Ampomah, E., Mensah, E., Gilbert, A. (2017), "Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages", Communications on Applied Electronics, vol. 7, no. 7, pp. 8–13.

[29] Palach, J. (2014), Parallel Programming with Python. Packt Publishing, Limited: Birmingham.

[30] Gehani, N. (1979), "A high level data structure - The grid", Computer Languages, vol. 4, no. 2, pp. 93–98.

[31] Schyberg, H., et al. (2020), Arctic regional reanalysis on single levels from 1991 to present. Copernicus Climate Change Service (C3S) Climate Data Store (CDS). Available online: https://cds.climate.copernicus.eu/cdsapp#!/dataset/reanalysis-carra-single-levels?tab=overview (accessed 31 January 2023).

[32] Koidan, K. (2021), Difference Between Python Modules, Packages, Libraries, and Frameworks. Available online: https://learnpython.com/blog/python-modules-packages-libraries-frameworks/ (accessed 14 February 2023).

[33] Cai, X., Langtangen, HP., Moe, H. (2005), "On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations", Scientific Programming, vol. 13, pp. 31–56.

[34] Unidata (n.d.), Network Common Data Form (NetCDF). Available online: https://www.unidata.ucar.edu/software/netcdf/ (accessed 12 May 2023).

[35] Teja, R. (2022), What Is NetCDF Data and Why Is It Interesting?. Available online: https://towardsdatascience.com/what-is-netcdf-data-and-why-is-it-interesting-ec26bcece19d (accessed 12 May 2023).

[36] Copernicus (n.d.), Climate reanalysis. Available online: https://climate.copernicus.eu/climate-reanalysis (accessed 16 May 2023).

[37] le Brocq, A. (2020), Ice sheet modelling. Available online: https://www.antarcticglaciers.org/glaciers-and-climate/numerical-ice-sheet-models/ice-sheet-modelling/ (accessed 24 April 2023).

[38] Vandecrux, B. (2022), GEUS-Glaciology-and-Climate/GEUS-SEB-firn-model: Python version of the GEUS SEB and firn model. Available online: https://github.com/GEUS-Glaciology-and-Climate/GEUS-SEB-firn-model (accessed 31 January 2023).

[39] Langen, PL., Mottram, RH., Christensen, JH., et al. (2015), "Quantifying Energy and Mass Fluxes Controlling Godthåbsfjord Freshwater Input in a 5-km Simulation (1991–2012)", Journal of Climate, vol. 28, no. 9, pp. 3694–3713.

[40] Abraham, N. (2015), Coding for dummies. 1st edition. John Wiley & Sons, Inc: Hoboken, New Jersey.

[41] Sanner, MF. (1999), "Python: a programming language for software integration and development", Journal of molecular graphics & modelling, vol. 17, no. 1, pp. 57–61.

[42] Kim, T., Cha, Y., Shin, B., Cha, B. (2021), "Survey and Performance Test of Python-based Libraries for Parallel Processing" in: The 9th International Conference on Smart Media and Applications (SMA 2020). Association for Computing Machinery: New York, pp. 154–157.

[43] Anderson, A. (2022), Understanding Computer Processors: CPUs vs. vCPUs and Threads vs. Cores. Available online: https://www.makeuseof.com/cpu-vs-vcpu-threads-vs-cores/ (accessed 26 April 2023).

[44] Qun, NH., Khalib, ZIA., Warip, MN., et al. (2016), "Hyper-threading technology: Not a good choice for speeding up CPU-bound code" in: 2016 3rd International Conference on Electronic Design (ICED). IEEE: Phuket, pp. 578–581.

[45] Amdahl, GM. (2013), "Computer Architecture and Amdahl's Law", Computer, vol. 46, no. 12, pp. 38–46.

[46] Cook, S. (2013), "Optimizing Your Application" In: Cook, S. (eds), CUDA Programming. Elsevier: Place of publication not identified, pp. 305–440.

[47] Ozgur, C., Colliau, T., Rogers, G., et al. (2017), "MatLab vs. Python vs. R", Journal of data science, vol. 15, no. 3, pp. 355–372.

[48] Stroustrup, B. (1986), "An overview of C++", ACM SIGPLAN notices, vol. 21, no. 10, pp. 7–18.

[49] McKinney, W. (2010), "Data Structures for Statistical Computing in Python", Proc. of the 9th Python in science conf. (Scipy 2010). Austin, Texas, pp. 56–61.

[50] Lindstrom, G. (2005), "Programming with Python", IT professional, vol. 7, no. 5, pp. 10–16.

[51] Gorelick, M., Ozsvald, I. (2014), High performance Python: practical performant programming for humans. First edition. O'Reilly: Sebastopol, California.

[52] NumPy (n.d.), NumPy - The fundamental package for scientific computing with Python. Available online: https://numpy.org/ (accessed 17 May 2023).

[53] Numba (n.d.), Numba - Numba makes Python code fast. Available online: https://numba.pydata.org/ (accessed 17 May 2023).

[54] pandas (n.d.), pandas. Available online: https://pandas.pydata.org/ (accessed 17 May 2023).

[55] Joblib (n.d.), Joblib: running Python functions as pipeline jobs. Available online: https://joblib.readthedocs.io/en/latest/# (accessed 17 May 2023).

[56] Yang, J., Xu, Y., Liu, Z. (2022), ERStruct: A Python Package for Inferring the Number of Top Principal Components from Whole Genome Sequencing Data.

[57] Anaya, M. (2020), Clean code in Python: develop maintainable and efficient code. Second edition. Packt: Birmingham, England.

[58] NumPy (n.d.), What is NumPy?. Available online: https://numpy.org/doc/stable/user/whatisnumpy.html (accessed 7 February 2023).

[59] van der Walt, S., Colbert, SC., Varoquaux, G. (2011), "The NumPy Array: A Structure for Efficient Numerical Computation", Computing in Science & Engineering, vol. 13, no. 2, pp. 22–30.

[60] Sloss, AN., Symes, D, Wright, C., Rayfield, J. (2004), "Chapter 12 - Caches" in: Sloss, AN., Symes, D, Wright, C., Rayfield, J., ARM System Developer’s Guide. Elsevier: Place of publication not identified, pp. 402–459.

[61] Numba (n.d.), A ~5 minute guide to Numba. Available online: https://numba.readthedocs.io/en/stable/user/5minguide.html#how-does-numba-work (accessed 7 February 2023).

[62] Numba (n.d.), Troubleshooting and tips. Available online: https://numba.pydata.org/numba-doc/latest/user/troubleshoot.html (accessed 10 March 2023).

[63] Numba (n.d.), Supported Python features. Available online: https://numba.pydata.org/numba-doc/dev/reference/pysupported.html (accessed 3 May 2023).

[64] Numba (n.d.), Performance Tips. Available online: https://numba.pydata.org/numba-doc/latest/user/performance-tips.html (accessed 10 March 2023).

[65] Numba (n.d.), Compiling Python code with @jit. Available online: https://numba.pydata.org/numba-doc/latest/user/jit.html (accessed 8 February 2023).

[66] IEEE SA (2019), IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic. Available online: https://standards.ieee.org/ieee/754/6210/ (accessed 15 March 2023).

[67] Saxena, P. (2022), How to Make Python Code Run Incredibly Fast. Available online: https://www.kdnuggets.com/2021/06/make-python-code-run-incredibly-fast.html (accessed 20 February 2023).

[68] Slatkin, B. (2019), "4. Comprehensions and Generators" in: Slatkin, B., Effective Python: 90 Specific Ways to Write Better Python, 2nd Edition. Addison-Wesley Professional: Place of publication not identified.

[69] Dang, AT. (2021), Here are some tips to speed up your Python program. Available online: https://levelup.gitconnected.com/here-are-some-tips-to-speed-up-your-python-program-e47257b4e6d3 (accessed 16 February 2023).

[70] Ratanghayra, A. (2021), 10 Ways To Speed Up Your Python Code!. Available online: https://medium.com/analytics-vidhya/10-ways-to-speed-up-your-python-code-bddd9f9902d0 (accessed 23 February 2023).

[71] McConnell, S. (2004), Code complete. 2nd ed. Microsoft Press: Redmond, Washington.

[72] Python Software Foundation (2023), The Python Profilers. Available online: https://docs.python.org/3/library/profile.html#module-cProfile (accessed 4 April 2023).

[73] Crall, J., et al. (2023), pyutils/line_profiler: Line-by-line profiling for Python. Available online: https://github.com/pyutils/line_profiler (accessed 4 April 2023).

[74] García, S., Luengo, J., Herrera, F. (2015), Data Preprocessing in Data Mining. Springer International Publishing: Cham.

[75] Fausto, RS., Van As, D., Mankoff, KD. (2022), AWS one boom tripod Edition 3. Available online: https://dataverse.geus.dk/dataset.xhtml?persistentId=doi:10.22008/FK2/8SS7EW (accessed 7 March 2023).

[76] Fausto, RS., Van As, D., Mankoff, KD., et al. (2021), "Programme for Monitoring of the Greenland Ice Sheet (PROMICE) automatic weather station data", Earth System Science Data, vol. 13, no. 8, pp. 3819–3845.

[77] Copernicus Climate Change Service, Climate Data Store (2023), ERA5 hourly data on single levels from 1940 to present. Available online: https://cds.climate.copernicus.eu/cdsapp#!/dataset/reanalysis-era5-single-levels?tab=overview (accessed 13 April 2023).

[78] NumPy (n.d.), numpy.clip. Available online: https://numpy.org/doc/stable/reference/generated/numpy.clip.html (accessed 15 March 2023).