Refactoring of Ocean Data Processing and Visualization Software Using Scientific Workflow Modeling

Eirin Sognnes

Master's thesis in Software Engineering

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 1, 2021

Contents

1 Introduction **7**
1.1 Underwater Acoustic Modeling 8
1.2 Scientific Workflows 10
1.3 DevOps 11
1.4 Design Science Research Methodology 12
1.5 Research Questions 13
1.6 Outline 14

2 Background and Related Work **16**
2.1 Acoustic Propagation Models 16
2.2 Scientific Workflows 21
2.3 Scientific Workflow Management Systems 23
2.4 Related Work 31

3 Problem Description and Analysis **35**
3.1 The Arctic Package 35
3.2 Software Architecture 36
3.3 Databases Used in the Arctic Package 39
3.4 Functionality and Requirements 41
3.5 The Arctic Package as a Scientific Workflow 46

4 KNIME Workflow Model **47**
4.1 KNIME Workflow Overview 47
4.2 Workflow Components 48
4.3 MATLAB Adaptations for the KNIME workflow 57

5 Airflow Workflow Model **61**
5.1 Airflow Workflow Overview 61
5.2 Workflow Operators 62
5.3 REST API 65
5.4 MATLAB Adaptations for the Airflow workflow 66

6 DevOps Pipeline **69**
6.1 Existing Arctic Package Pipeline 69
6.2 Updated Pipeline 69

7 Generated User Interfaces **80**
7.1 Original GUI 80
7.2 KNIME Built-in GUI 84

7.3 React JSON Schema Form . . . 88

8 Evaluation **96**
8.1 Criteria for Evaluation . . . 97
8.2 Software Quality Evaluation . . . 99
8.3 Reliability . . . 100
8.4 Usability . . . 103
8.5 Portability . . . 112
8.6 Maintainability . . . 115
8.7 Compatibility . . . 124
8.8 Functional Suitability . . . 125
8.9 Performance Efficiency . . . 126
8.10 Evaluation Summary . . . 136

9 Conclusions and Future Work **138**
9.1 Research Questions Revisited . . . 138
9.2 Threats to Validity . . . 143
9.3 Future Work . . . 144

Glossary **155**

Acronyms **156**

A MATLAB Log **164**

Acknowledgements

Throughout the work with this master thesis I have received valuable guidance and support. First, I wish to thank my supervisor, Lars Michael Kristensen for assistance and feedback at every stage in this master project. I also wish to thank Espen Storheim, my contact at NERSC who provided information and insight into the Arctic Package and the acoustics domain. Additionally, I would like to thank Rogardt Heldal, Hanne Sagen, and Kjell Eivind Frøysa who took part in shaping this master thesis.

Abstract

The Arctic Package is a MATLAB software package for modeling and visualization of acoustic propagation. It aims to improve the accessibility and understanding of ocean processes and how they affect the climate. This thesis investigates how the Arctic Package can be adapted to a scientific workflow context and how the adaptation affects software quality. Two workflow models were created, one with a textual specification implemented in Airflow and one with a graphical specification implemented in KNIME. To adapt the Arctic Package to a scientific workflow, the package was refactored and a new graphical user interface created. Additionally, a DevOps pipeline was established to simplify distribution and setup of the workflows. The software quality for the original Arctic Package and the two workflow models was evaluated using the ISO/IEC 25010 standard. The results showed that the Airflow workflow scored best in total for software quality, but overall software quality improved in both of the workflow based implementations compared to the original Arctic Package.

Chapter 1

Introduction

The climate is changing, especially in the Arctic region. The Coordinated Arctic Acoustic Thermometry Experiment (CAATEX) project collects data from under the sea ice, an area that remains mostly unknown [1]. The CAATEX project is lead by the Nansen Environmental and Remote Sensing Center (NERSC), a non-profit research foundation that was founded in 1986 [2]. NERSC has seven research groups, and the CAATEX project is undertaken by the Acoustics and Oceanography group [3].

To collect data from under the sea ice, CAATEX uses acoustic thermometry and standard oceanographic instrumentation. Acoustic thermometry, or acoustic tomography, is a technique where sound signals are transmitted from a source to a receiver [4]. The average temperature along the path of the sound is inverted from the average sound speed, which is computed from the travel time and the distance that the sound has propagated. Sound travels at approximately 1500 m/s in the ocean [5]. Consequently, one measurement along a given track would take a fraction of the time compared to use of ships or gliders. In addition, measurements can be made in ocean beneath the sea-ice, where one normally would need an icebreaker to go.

An important component in acoustic tomography is underwater acoustic modeling, which describes how sound propagates in the ocean. This is used in the planning phase to set up experiments, and in the analysis of the results to identify different arrivals. There are, however, many other applications of underwater acoustic modeling, e.g., acoustical oceanography [6], [7], acoustic communication [8], and investigation of man-made noise on marine wildlife [9]. In many cases they are used in comparison with experimental data to get a better understanding of the results, or to validate the models. Such models have had a significant gain in popularity in the 1980's and 1990's with the advances in computer technology. Larger and more complex problems can now be solved, such as full 3D modeling [10]–[12]. However, the use of these models is not always straight forward due to a lack of documentation, and complicated input formats.

Acoustic models require information about the properties of the ocean. Such data is sparse in the Arctic. The sea-ice cover in the Arctic Ocean limits the ship traffic and use of other observation methods, such as moorings or gliders. Thus, when modeling large-scale or time-dependent propagation, one must typically rely on either ocean models or databases that are constructed from a series of measurements. Download-

ing and processing such data manually can be complicated and time-consuming.

The Arctic Package [13] is a software package that can model and visualize the propagation of sound in the ocean between a source that transmits an acoustic signal and a receiver that captures the signal. It was created to simplify the acoustic modeling by automatically fetching the necessary input parameters, such as sound speed and bathymetry, setting up the specific input files, and visualizing the results from the simulation. Currently, it consists of acoustic models implemented in Fortran that can be run from a Graphical User Interface (GUI) implemented in MATLAB. Furthermore, there are MATLAB scripts to prepare the input files for the model and plot the results from the runs of the models. The simulation models that are available in the package are Bellhop [14], Eigenray [15], KRAKEN [16], MPIRAM [17], and RAM [18].

Today the Arctic Package is only used to a limited extent by the researchers at NERSC. This is a result of a complicated GUI that can be difficult to understand and time consuming to use. Two former master students, Van den Bergh [19] and Klockmann [20], changed the GUI in order to improve the usability and extended the set of available simulation models. However, there are still challenges with the Arctic Package, including limited automation and difficulty to run multiple simulations consecutively or in parallel. Furthermore, there are challenges related to maintenance of the code, such as tight coupling between the MATLAB scripts and the GUI. Another challenge in the Arctic Package, is to set up the software on a new computer. The acoustic models are implemented in Fortran and a new user must be able to compile and run the Fortran source code in addition to the MATLAB scripts. This is further complicated by the lack of instructions on how to set up and use the package.

The rest of the chapter is organized as follows: Section 1.1 gives a general overview of acoustic modeling, Section 1.2 introduces scientific workflows, Section 1.3 introduces the concept of Development (Dev) and Operations (Ops), Section 1.4 describes the scientific method used in this thesis in the form of Design Science Research (DSR), Section 1.5 introduces the research questions, and Section 1.6 describes how the rest of this master thesis is structured.

1.1 Underwater Acoustic Modeling

Underwater acoustic modeling is concerned with how sound propagates through water. To create accurate models of how the sound moves, factors such as temperature, salinity, and pressure needs to be considered [21]. As sound moves through the ocean, it interacts with the environment and effects such as reflection, refraction (bending), and scattering occurs. Refraction of sound occurs when the sound speed changes, as the sound bends towards lower speeds. Reflection generally occurs at the sea floor or at the sea surface. When sea ice is present it usually results in a higher loss of sound than without ice. This is due to the rough underside of the ice which scatters the sound in all directions. Consequently, the sound will propagate a shorter distance when ice is present.

In the Arctic Package, the acoustic models require information about the bathymetry, the sound speed, the source that sends the acoustic signal, and the receiver that

receives the acoustic signal. Furthermore, the models require information about the distance from the source to the receiver.

1.1.1 Sound Speed

The speed of sound in the ocean is a function of temperature, salinity and pressure [22]. When the temperature, salinity, or pressure increases, the sound speed also increases. Temperature is the promoter that has the most impact, where a 1°C change in temperature corresponds to about a 4 m/s change in sound speed [5].

A sound speed profile shows how the sound speed changes with depth. An illustration of the general sound speed profiles in the ocean is shown in Figure 1.1. The solid line represent a generic sound speed profile. In the Arctic, the sound speed acts different from the generic sound speed profile, due to the cold climate. The dotted line to the left in the figure, represent a sound speed profile in the Arctic.

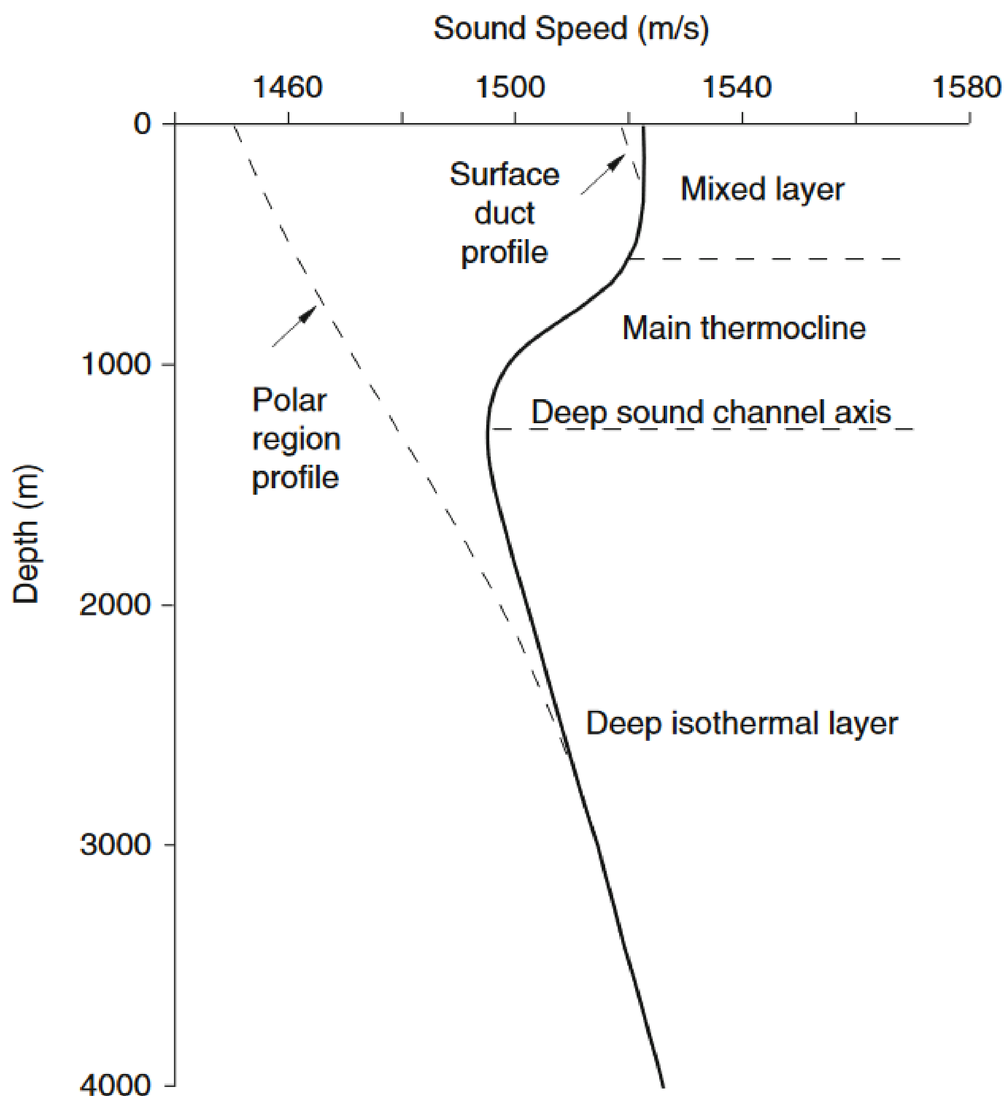


Figure 1.1: Illustration of generic sound speed profiles as a function of depth [22]

The surface areas are often mixed because of wind and waves, illustrated by the mixed layer in Figure 1.1. In the mixed layer the temperature is constant. The top of Figure 1.1, shows a surface duct profile by a dotted line. The surface duct profile has a slightly increasing sound speed with depth, because of the increasing pressure and the constant temperature. This causes sound rays to bend upwards to the surface, where they are reflected back down, trapping the rays in the surface duct.

Underneath the mixed layer in Figure 1.1 is the main thermocline, where the temperature decreases with depth until it reaches a temperature of about 2°C [22]. The Deep Sound Channel (DSC) axis is where the thermocline ends and the sound speed is at its lowest. As in the surface duct, the sound can be trapped in the DSC axis. Underneath the thermocline is the deep isothermal layer (at the bottom of Figure 1.1), where the temperature is constant, but the pressure increases and thus the sound speed increases.

In the Arctic, the water is at its coldest close to the surface and the minimum sound speed is also close to the surface. In Figure 1.1, the dotted line has an increase in sound speed throughout the profile, due to increasing pressure. At what would be the thermocline in a non-Arctic area, the temperature is increasing in the Arctic, which contributes to increasing the sound speed.

1.1.2 Bathymetry

Bathymetry is the study of the beds of water bodies, such as the seabed. The data about the bathymetry used in the Arctic Package is bathymetry data for latitudes higher than 64° North stored in The International Bathymetric Chart of the Arctic Ocean (IBCAO) database [23]. The current version, 3.0, has a coverage of the area from 64° North of about 11%. This means that large areas are still unknown.

1.2 Scientific Workflows

Scientific workflows [24] represent the steps in the process of a scientific analysis, such as collecting data and executing analysis and/or simulation tasks on the collected data. It is used to improve a scientific process and can be used to automate the steps in a scientific analysis. These steps can be collecting data, preparing data, modeling data, and plotting results. A scientific workflow can also help monitoring and managing the data in a workflow.

Scientific workflows are typically visualized as graphs. The nodes in the graphs represent tasks and resources, while the connections in the graphs represents the dependencies between the tasks. In a scientific workflow, tasks define the operations to be performed. A basic example of a workflow is shown in Figure 1.2. A scientific workflow is built from three basic components: the input data to the workflow, the computational tasks performed on the data, and the dependencies between the tasks [25]. In Figure 1.2, the input data is read in the `File Reader` node at the start of the workflow. The connections are the arrows between the nodes, which indicates the dependencies. The `Row Filter` depends on the `Column Filter`, which depends on the `File Reader`. Each node in the workflow corresponds to a computation. The

first node (orange) reads the input file, the second node filters on columns, the third node filters on rows, and the remaining (blue) nodes performs plotting operations.

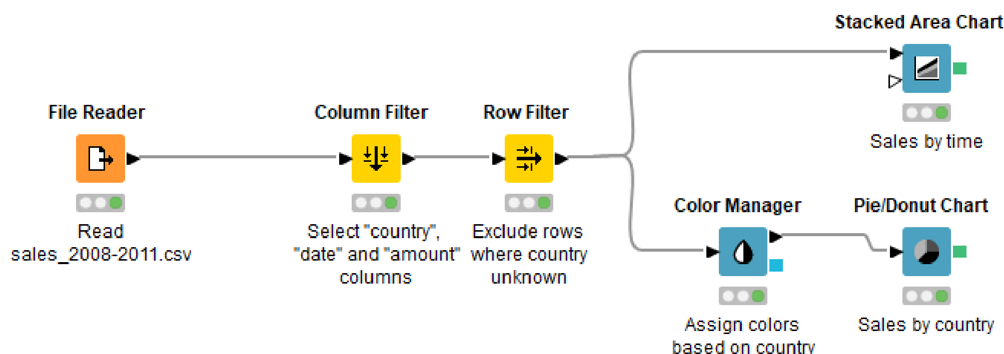


Figure 1.2: A workflow example for processing a sales dataset created in KNIME. The grey rectangular bar below the nodes indicates current execution status.

The two main approaches to create workflows are graphical and textual. The graphical approach uses blocks to represent tasks in the workflow, and the arcs between them to represent the dependencies. The graphical approach can ease creation and execution of workflows for scientists that have limited experience with programming. This is due to the graphical representation of tasks and dependencies, which abstract away the underlying code that executes the computational steps. The textual approach uses a domain-specific language to arrange and specify the scientific workflows. This typically requires more knowledge about programming and can be harder for the scientist to set up and use.

Both textually and graphically specified workflows are generally visualized as Directed Acyclic Graphs (DAG). The DAG workflow models can contribute to a better understanding of the processes in the system. In the sales dataset example (Figure 1.2), the graph shows that the dataset is processed by selecting the country, date, and amount columns and then filtering out rows with unknown country. This makes the processing transparent, and the user knows exactly how the information was processed to produce the visualizations of the sales dataset.

1.3 DevOps

DevOps is a combination of practices from development (Dev) and operations (Ops) for IT systems [26]. The purpose of DevOps is to enable fast delivery from software development to deployment of a new version of the software. This is achieved through a DevOps pipeline that automates development and deployment. A DevOps pipeline typically consists of Continuous Integration (CI) and Continuous Delivery/Deployment (CD). CI is the practice of integrating code changes from developers into a shared code repository. When the code is merged, it is normally

automatically built and tested. CD is the practice of deploying the code changes to a production environment. When CD is automatic it is called continuous deployment and when it is manual it is called continuous delivery.

1.4 Design Science Research Methodology

Design science research (DSR) is the methodology used in this thesis. DSR focuses on the artifact that is studied and describes how to approach design of the artifact in its context [27]. The artifacts of DSR are intended to solve or improve a problem in the context, which means that the stakeholders are important. In the context of this thesis, the artifact is the Arctic Package, and the stakeholders are the researchers at NERSC, who use the software.

DSR is similar to design, but instead of just focusing on the artifact, DSR aims to be relevant for both the immediate stakeholders and for the research community in general [28]. This means that the thesis must relate the results to existing research and make them available to the stakeholders and researchers.

There are five main activities in DSR [28]:

A1 **Explicate the problem** - To investigate and analyze a problem and its causes.

A2 **Define the requirements** - To find a solution to the problem and how to develop the artifact to solve it.

A3 **Design and develop the artifact** - To create the artifact based on the requirements.

A4 **Demonstrate artifact** - To use it in a real-world situation to demonstrate that it solves or improves on the problem.

A5 **Evaluate artifact** - To determine how well the solution works and fulfills the requirements.

When performing the activities, they must be connected to relevant knowledge and the artifact must be developed in a way that ensures that it relevant to the stakeholders. The three cycles of DSR in Figure 1.3, addresses how the activities can be related to both the scientific knowledge base and to the environment where the artifact is used.

The three DSR cycles are the relevance cycle, the design cycle, and the rigor cycle [29]. The relevance cycle connects the environment to the project. The requirements must be developed with the stakeholders and evaluated with regards to the requirements in collaboration with the stakeholders. The requirements for the Arctic Package (Section 3.4) are based on conversations with the stakeholders at NERSC and an analysis of the existing Arctic Package. The rigor cycle connects the project to the knowledge base. In this case, scientific workflows form the basis for the rigor cycle and this master thesis uses the knowledge from related work (Section 2.4) as a foundation. The rigor cycle also includes the research contributions from this thesis to the knowledge base.

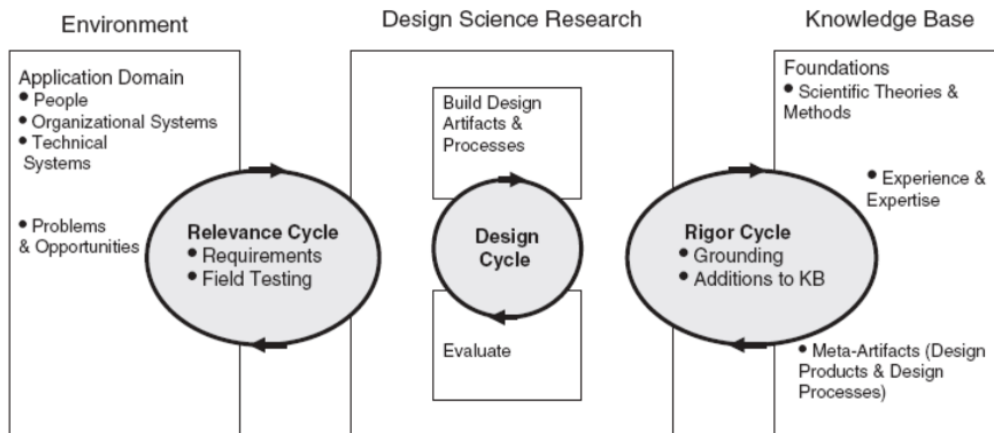


Figure 1.3: Design science research cycles by Hevner [29]

The design cycle is concerned with the development of the artifact, and iterates between construction, evaluation, and refinement. The artifacts developed for this project are the two workflows. They were developed based on input from the relevance cycle and the rigor cycle, where the relevance cycle contributed with the requirements for the application, and the rigor cycle constituted the theoretic foundation for selecting the scientific workflow management systems and provided insight into existing scientific workflows used during design and development of the workflows.

1.5 Research Questions

To improve the Arctic Package, this master thesis explores how a combination of scientific workflows, container technology, and automatic code generation can contribute to improve the software quality of the Arctic Package. In order to create a scientific workflow model, the requirements for the Arctic Package must be identified, and based on those a workflow model can be created. The workflow model requires information about the data the package uses and the parameters the user provides. Furthermore, it is necessary to identify the tasks performed by the different parts of the Arctic Package and which order they are performed in, i.e., the dependencies between tasks. The dependencies also specify which tasks that can be performed in parallel.

The research questions investigated in this thesis are:

RQ1 What kinds of scientific workflow management systems exist, and which ones are suited for the Arctic Package?

RQ2 How can the computations performed by the Arctic Package be represented by a scientific workflow model? What kind of data and which tasks are required in a scientific workflow model of the Arctic Package?

RQ3 Can a graphical user interface for the Arctic Package be automatically generated based on a specification of the Arctic Package input parameters?

RQ4 How does a DevOps pipeline combined with container technology affect software quality in the context of the Arctic Package?

RQ5 How does scientific workflows affect software quality in the Arctic Package?

With regards to the five activities in DSR, RQ1, RQ2, and RQ3 contribute to explicating the problem and to define the requirements for the artifact. Explication is concerned with describing the data and tasks in the Arctic Package, identifying what needs to be improved upon, and understanding how a scientific workflow may contribute to achieve an improvement. Defining the requirements is part of determining how the computations can be represented as a scientific workflow. These requirements must be identified in collaboration with the stakeholders, who can explain how they use the package and what functionality they need.

RQ4 and RQ5 contribute to evaluating the artifact. They investigate how the changes to the Arctic Package has affected different aspects of it. RQ4 evaluates how the changes to the DevOps pipeline affects the quality, while RQ5 evaluates how the scientific workflow affects software quality and how the different versions of the package compares.

Design, development, and demonstration is not explicitly part of the research questions, but instead a necessary step to go from design to evaluation. If the artifact is not developed and tested, there would not be an artifact to evaluate. Hence, activities 3 and 4 of the DSR are implicitly carried out through the research questions.

1.6 Outline

The rest of the thesis is organized into eight chapters.

Chapter 2 - Background and Related Work describes the background for this thesis. It introduces the acoustic propagation models that are part of the Arctic Package, describes scientific workflows, and surveys different scientific workflow management systems. Furthermore, Chapter 2 describes related work to the Arctic Package and to the scientific workflow domain.

Chapter 3 - Problem Description and Analysis describes how the Arctic Package is structured, what the goal for the package is, and its requirements. Additionally, it investigates how the Arctic Package can be adapted to a scientific workflow context.

Chapter 4 - KNIME Workflow Model describes the KNIME workflow model that was developed for the Arctic Package. The chapter contains details about the nodes used in the workflow, and which adaptions that were done to transform the Arctic Package into a KNIME workflow.

Chapter 5 - Airflow Workflow Model describes the Airflow workflow model that was developed for the Arctic Package. The chapter describes the workflow operators, the REST API that Airflow includes, and the MATLAB adaptations needed to transform the Arctic Package into an Airflow workflow.

Chapter 6 - DevOps Pipeline contains details about the pipeline for the original Arctic Package, and explains the changes to the pipeline for the workflow-based versions. For each of the workflow implementations for the Arctic Package, the tools used in the updated pipeline, the setup, and the process to change the workflow, are outlined.

Chapter 7 - Generated User Interfaces details the different components in the original GUI, the KNIME GUI, and the Airflow GUI. The chapter includes information about the structure, mechanisms in the GUIs, and how the GUI communicates with the Arctic Package.

Chapter 8 - Evaluation introduces software quality models and the quality criteria that are relevant with regards to evaluating the Arctic Package. The criteria are then broken down into sub-criteria that are used to evaluate how well the different versions of the Arctic Package satisfy the quality criteria and the requirements for the package.

Chapter 9 - Conclusion and Future Work summarizes the project and relates the findings back to the research questions and methodology. Additionally, Chapter 9 describes future work that is relevant for the Arctic Package and threats to validity.

Chapter 2

Background and Related Work

In this chapter, background information about scientific workflows and the acoustic models are explained. Section 2.1 gives a description of the acoustic models used in the Arctic Package, Section 2.2 describes different types of scientific workflows, Section 2.3 goes into detail on the two Scientific Workflow Management Systems (SWFMS) chosen for this master project, and Section 2.4 describes related work on scientific workflows and the Arctic Package.

2.1 Acoustic Propagation Models

Acoustic models are used to predict how the sound propagates through the ocean. As described in Section 1.1, the propagation is affected by temperature, salinity, and pressure, which causes the sound to propagate differently based on changes in those factors. The environment in the sea, such as the sea floor and sea surface, also affects the propagation. The acoustic models used in the Arctic Package are Bellhop, Eigenray, KRAKEN, MPIRAM, and RAM. Eigenray and Bellhop are ray models, RAM and MPIRAM are Parabolic Equation (PE) methods, and KRAKEN is a normal mode program. The three modeling categories are explained below.

Ray models use ray tracing to calculate the sound path through the ocean. Ray tracing is about understanding how rays bend and refract. To do ray tracing, a field of rays is generated. Each ray is normal to a wave front, where the wave front is a surface consisting of all points on a wave at the same location. Parabolic equation methods are used for solving range dependent problems. The parabolic equation method bases itself on solving equations derived from the Helmholtz equation using parabolic approximations [22]. Normal mode methods solves an equation dependent on depth. The acoustic field is created by summing up the contributions of each mode, where the weight of each mode is relative to the source depth [22].

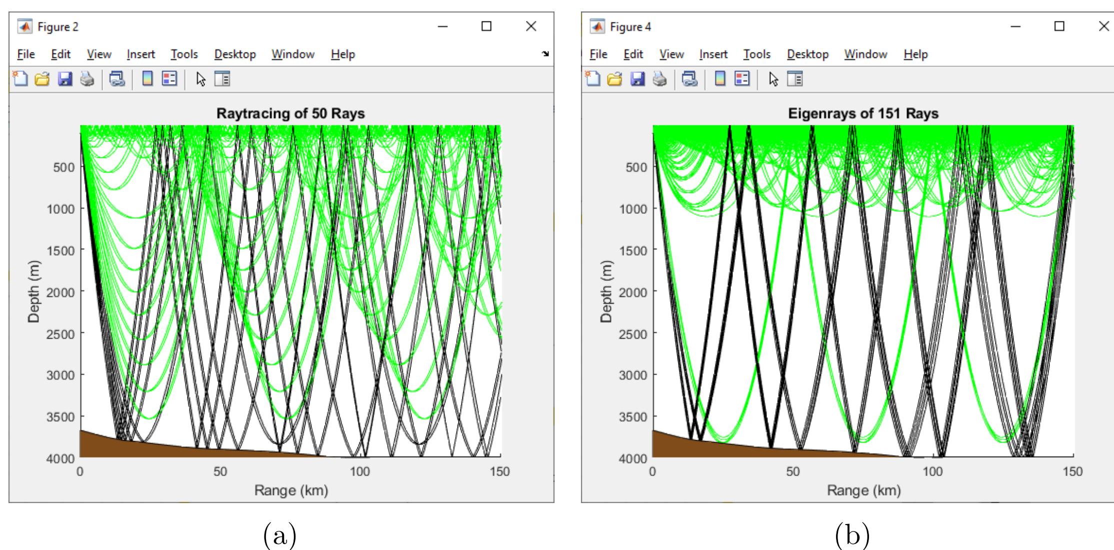
2.1.1 Bellhop

Bellhop is implemented by Porter [14], and is a model for ray tracing and simulation of acoustic pressure fields in ocean environments. This master project uses the Fortran implementation, but Bellhop has also been implemented in MATLAB and Python. Bellhop can be used with five different options: ray tracing, eigenrays,

and three different transmission loss options: coherent, semi-coherent, and incoherent.

Bellhop uses sound speed profiles or sound speed profile fields to perform ray tracing. The Bellhop model has options to specify files containing bathymetry and sound speed information for range dependent calculations. Other options in the Bellhop model, are the bottom reflection coefficient, the surface reflection coefficient, and the source beam pattern. The input file for Bellhop (an environment file with a .env file extension) specifies the options, files, and parameters that the Fortran model uses. Examples of such parameters are frequency, source and receiver depth, and the number of rays. These parameters are specified by the user in the Arctic Package.

Figure 2.1 shows the resulting plots of a Bellhop run with the ray tracing option (left) and the eigenray option (right). The ray tracing option uses ray equations to calculate the coordinates of the rays. The rays are frequency independent, and therefore the frequency parameter is not important for the ray tracing and eigenray options [14]. For the number of rays parameter, Porter recommends using about 50 rays for a ray trace run of Bellhop, to avoid getting a plot that is hard to read. In the Arctic Package, the number of rays parameter has a default value of 50. The output from the Bellhop model is a .prt file and, depending on the model type, either a .ray or a .shd file. The .ray file is produced for eigenray calculations or ray tracing and contains the information about the calculated rays. The .shd file is produced if any of the transmission loss options were chosen. The .prt file contains the log from the simulation and is used to verify the model setup and diagnose any errors.



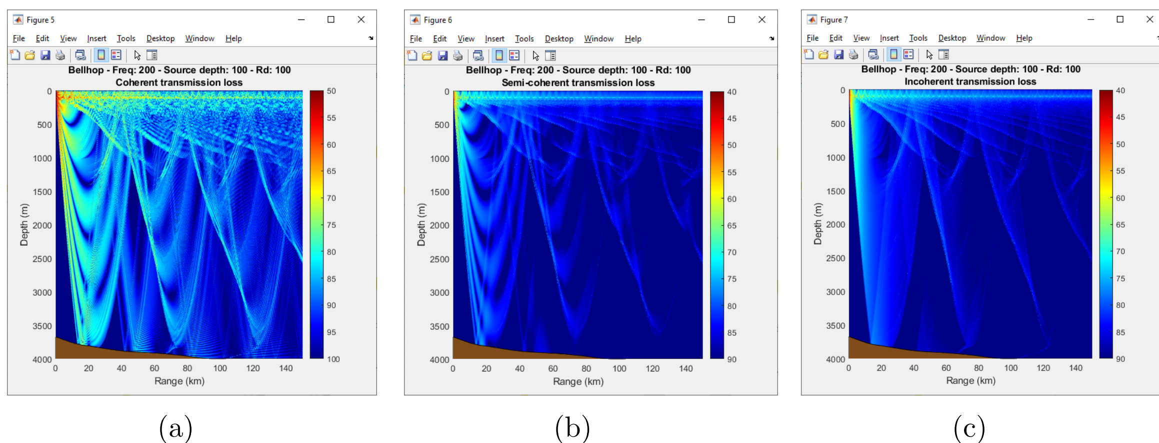
(a) (b)

Figure 2.1: Ray paths computed with Bellhop for a ray simulation (a) and an eigenray simulation (b). The black lines hit the surface and the bottom and the green lines only hits the surface. Blue lines occur if only the bottom was hit, and red lines if neither the surface nor the bottom was hit.

The Bellhop eigenray option does the same calculations as the ray tracing option, but only saves the rays that connects a source to a receiver, called eigenrays. As some of the rays miss the receiver, the parameter determining the number of rays should be set to a higher number than for a normal raytracing. In the Arctic Package

it has a default value of 501.

When computing transmission loss, the Bellhop model calculates the pressure field for the receivers. In this instance, the frequency parameter is important, as the interference pattern is directly related to the wavelength [14]. There are three types of transmission loss: coherent, semi-coherent, and incoherent. The plots of the three transmission loss options are shown in Figure 2.2. The coherent transmission loss preserves all details of the interference pattern. If the details are not necessary, it is possible to compute the incoherent transmission loss. If the user wants to preserve some detail, such as if the interference effects are not significant or reliably predictable, it is best to compute the semi-coherent transmission loss [14].



(a) (b) (c)

Figure 2.2: Transmission loss computed with Bellhop as a function of range and depth for a source at 100 m radiating with 200 Hz frequency. (a) shows the coherent transmission loss, (b) the semi-coherent transmission loss, and (c) the incoherent transmission loss.

2.1.2 Eigenray

The Eigenray model has the option to calculate eigenrays or timefronts. Eigenrays are the rays that connect a source to a reveiver. Timefronts are the ray depth versus travel time for a fixed range [30]. The Eigenray model has the option to run for eigenrays or for timefronts. It is based on the RAY code and technical report of Bowlin et al. [31]. The RAY model is implemented in C and the Eigenray model is implemented in Fortran. The Eigenray code is optimized for long-range, deep water calculations, with the goal of achieving fast and accurate predictions of the wavefronts and eigenray travel times at basin scale [15]. It uses look-up tables for sound speed, the sound speed gradient, and other parameters, using depth as index. To keep the computation fast, the step size is predetermined. The alternative is adaptive stepping, which recalculates the step size with every step. Predetermined step size is possible, as the step size is already known to be within a certain interval.

When running the Eigenray model for eigenrays, the results are visualized by the two plots shown in Figure 2.3. The top plot shows the arrival times and arrival angle of the rays, while the bottom plot shows the ray paths of the eigenrays (light blue) along with the bathymetry (darker blue line). When running the Eigenray

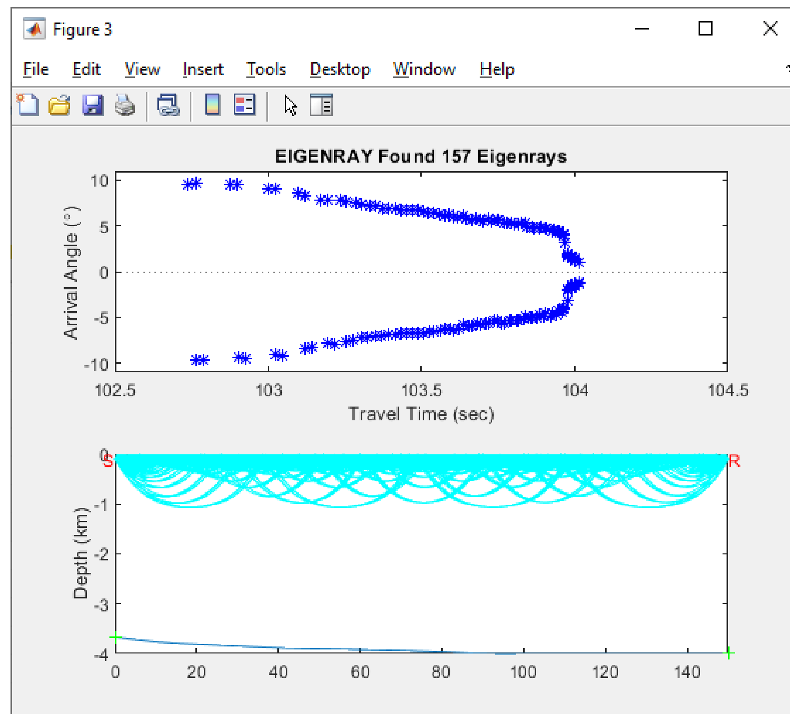


Figure 2.3: Results calculated with Eigenray for a source at 100 m depth and a receiver at 100 m depth. The top plot shows the arrival angle of the eigenrays. The bottom plot shows the path the eigenrays traveled from source to receiver in light blue. The darker blue line shows the bathymetry.

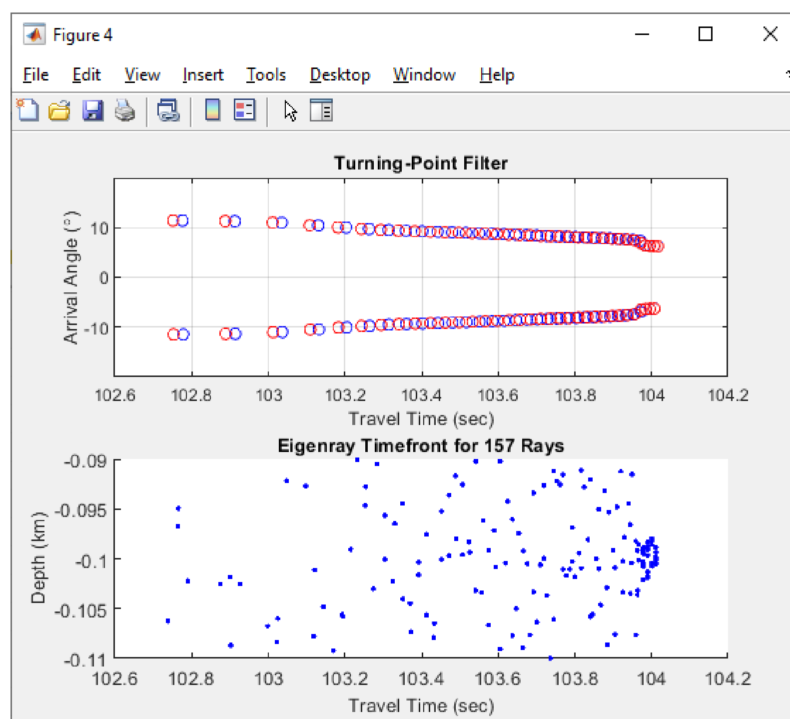


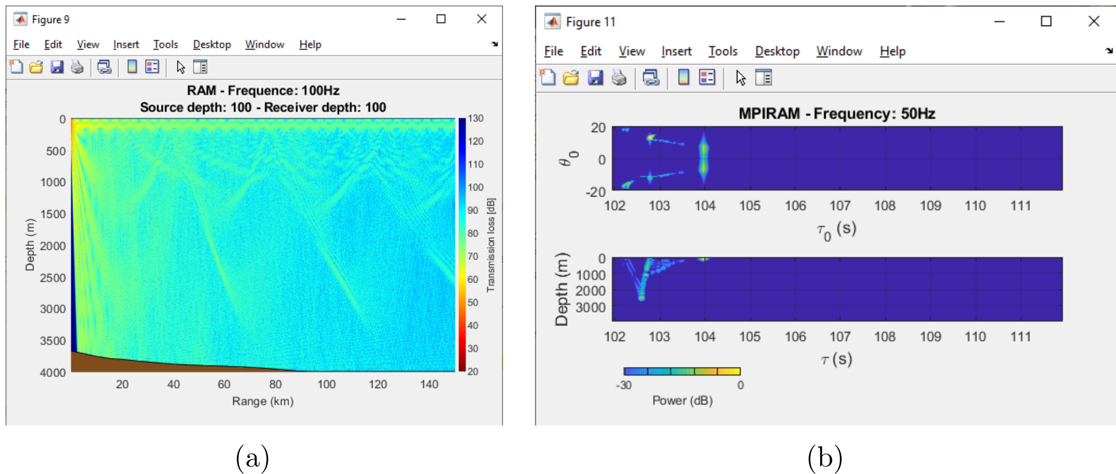
Figure 2.4: Results calculated with Eigenray timefronts for a source at 100 m depth and a receiver at 100 m depth. The top plot shows the turning point filter and the bottom plot shows the timefronts of the eigenrays.

model for timefronts, the plot shown in Figure 2.4 is created. The top plot shows the turning-point filter while the botton plot shows the timefronts. The turning point of a ray is where its angle is zero [30]. The turning point plot shows the turning point of the ray with regards to travel time. It also shows the angle the sound is received at by the receiver.

2.1.3 Range Dependent Acoustic Model (RAM)

The Range Dependent Acoustic Model (RAM) is based on the Parabolic Equation (PE) method, specifically the split-step Padé solution, and calculates the frequency-dependent wave propagation through the ocean [18]. The PE method uses the assumption that outgoing energies dominate backscattered energy. The split-step Padé solution is faster and has the same accuracy as other PE solutions, which yields an improved efficiency [32]. It is useful for modeling sound propagation in the ocean, as it is effective while still allowing wide propagation angles, large ranges, and large depths.

There are two different versions of RAM in use in the Arctic Package: RAM and MPIRAM. RAM was implemented by Collins [32], while MPIRAM was implemented by Dushaw [17]. RAM computes and writes to file the transmission loss on the path from source to receiver. The transmission loss is a measure of the decrease in intensity of the sound. RAM also computes and writes to file a range-depth grid. MPIRAM uses the Message Passing Interface (MPI) system to create a parallelized version of RAM. While RAM calculates the transmission loss, MPIRAM calculates the timefronts and turning points. Figure 2.5 shows the output plots of RAM (2.5a) and MPIRAM (2.5b).



(a) (b)

Figure 2.5: Results calculated with the two different RAM models. The transmission loss versus range and depth is shown in (a) for a source at 100 m vibrating at 100 Hz. The turning-point filter and timefronts calculated with MPIRAM are shown in (b) for a source at 100 m vibrating at 50 Hz.

2.1.4 KRAKEN

KRAKEN is a normal mode model [16]. Figure 2.6 a visualization of the results from running the KRAKEN model. The top plot shows the turning point filter, the middle plot shows the KRAKEN timefronts, and the bottom plot shows the modes on top of a bathymetry and the annual average of the sound speed profiles. In the original version of the Arctic Package, the KRAKEN model uses a Matlab EXecutable (MEX) wrapper to make it easier to run the model from MATLAB. The MEX wrapper invokes a C program that runs the KRAKEN model implemented in Fortran.

Figure 2.6: Plot of the KRAKEN model results. The top plot shows the turning point filter, the middle plot shows the timefronts, and the bottom plot shows the normal modes on top of the bathymetry (blue line) and sound speed profile (red line) plot.

2.2 Scientific Workflows

Scientific workflows represent the steps in the process of a scientific analysis. Early scientific workflows were implemented as shell scripts, which has a high complexity and is difficult to use for non-programmers. The scripts often involves a lot of

preprocessing of the data, in order to get it to a structure readable for the analysis tool, and then a lot of postprocessing to make the output readable to a human [33]. With the scripting approach, there are challenges regarding data management, code portability, and scaling. SWFMSs were created to handle these issues. The SWFMSs are either graphical tools or textual. KNIME is an example of a SWFMS that uses a graphical workflow representation, and the sample workflow introduced in Section 1.2 and shown in Figure 1.2 was created in KNIME.

Scientific workflows aim to simplify common processes, such as collecting, cleaning, and analyzing data. Moreover, they can contribute to documentation, as organizing and dividing a larger task into subtasks for a scientific workflow makes the process more transparent. Vigder et al. [34] found that scientific workflows created in Sweet, a SWFMS they developed, improved productivity through ease of use. This was a result of the users not having to learn the scripting language used to create the workflows, and that parameters could be abstracted away from the code. Another paper by Radosevic et al. [35] found that scientific workflows through KNIME helped make environmental modeling more reproducible and transparent.

2.2.1 Control- and Data-Driven Workflows

Workflows can be divided into two main groups: control-driven and data-driven workflows [36]. Both groups has tasks and dependencies in the form of connections between the tasks, but the connection has different purposes. Control-driven workflows focus on the order of execution, and are most common in business workflows [37]. This means that the connection represent the order the tasks are executed in. Data-driven workflows focus on the transformation of data, and consequently the connections represent data dependencies. Scientific workflows often have properties of both control- and data-driven workflows, as they need to address both computational and data aspects of a research data analysis process [38].

Control-Driven Workflows

A control-driven workflow consists of a sequence of operations and control structures to control the flow between components [36]. In a graphical SWFMS there are tasks and task dependencies, where the tasks are represented by nodes and the task dependencies are represented as edges between the nodes. In a control-driven workflow, data does not flow through the edges between the tasks, as it would in data-driven workflows. Instead, the dependencies represent the order in which tasks are run in. Figure 2.7 shows an example of a control driven workflow, implemented in Airflow. In the workflow, the edges specify which tasks run when. For example the task `get_config` needs to finish before any other task starts, but `create_map` and `prepare_input` can run at the same time, as they are not connected.

Some languages also support the use of conditionals, such as if-statements and for-loops. These structures are inherently control-driven, as they specify what task to execute (if) or how many times to execute a certain task (for), and nothing about the data-dependencies between the tasks. Shields [36] argues that if support for many programming constructs, such as ifs, are included, it will counteract the simplicity the workflows provides, and risks becoming another high-level programming language.

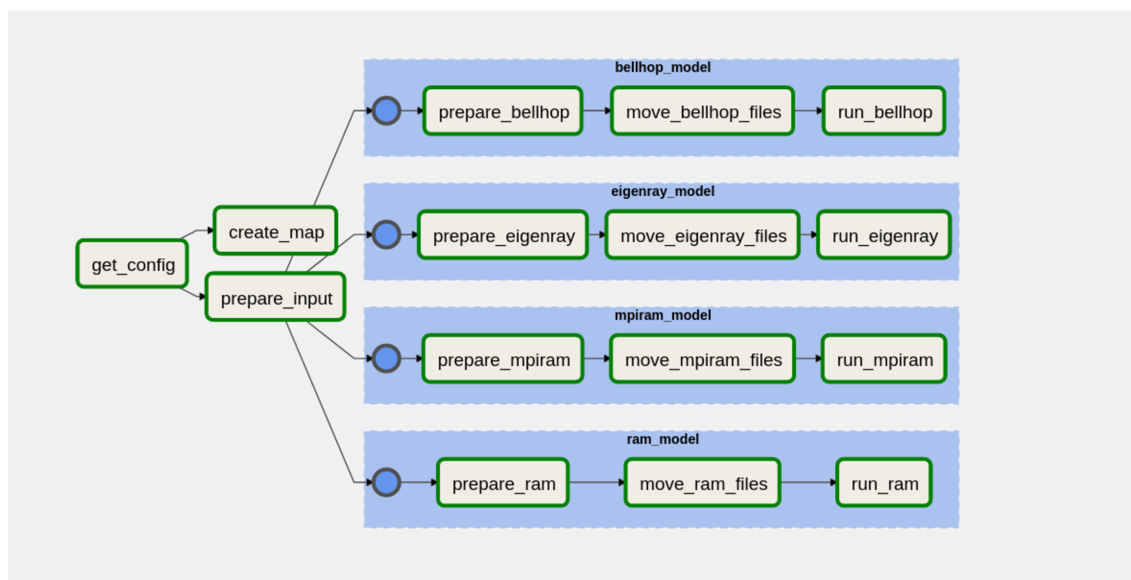


Figure 2.7: A visualization of the DAG in an Apache Airflow workflow. The tasks do not transfer data between each other, but if two tasks are connected by an edge, the task to the right cannot run until the task to the left has completed.

Data-Driven Workflows

Data-driven workflows are generally simpler than control-driven workflows, because they do not contain control functions [36]. Instead, they consist of tasks and dependencies, and some of the tasks can have implicit control-functions. For example an if-statement can be replaced by a component with two outputs. The two outputs would be different based on a condition, and then the if-statement is not needed. The data-driven workflow representation is focused on the data that goes into the component and the data product, instead of controlling the workflow through if-statements and other control functions. This means that which tasks are executed in sequence and in parallel is not explicitly defined, but inferred from the data dependencies [39].

2.3 Scientific Workflow Management Systems

There are numerous scientific workflow management systems such as Kepler [40], Pegasus [41], Apache Taverna [42], Apache Airavata [43], Apache Airflow [44], Prefect [45], Galaxy [46], and KNIME [47]. An element that separates scientific workflows from scripting approaches to platform and tool automation, is that SWFMSs are typically based on dataflow languages as opposed to imperative languages [48]. They also provide documentation, abstraction, workflow execution monitoring, and workflow optimization.

An example of an abstraction that can be made in scientific workflows, is a component in KNIME. Components allow workflows to group a set of nodes together and represent them by a parent node. This makes the workflow less complex, as small related tasks can be grouped together in a component node. This concept is further described in Section 2.3.1.

The SWFMSs target different domains and also use different approaches to create workflows. Galaxy and Taverna are aimed toward the bioinformatics domain and genomics analysis. Others, such as Airavata, Airflow, Kepler, KNIME, Pegasus, and Prefect are general purpose. The tools employ different strategies to create workflows. Airflow, Pegasus, and Prefect use textual languages, while Kepler and KNIME use graphical user interfaces. Others, such as Airavata and Taverna use both.

For this master project one graphical and one textual management system are used to build a workflow for the Arctic Package. The selected workflow management systems are KNIME for the graphical workflow model and Airflow for the textual workflow model. KNIME was chosen as the graphical tool because it is:

- Open source and free to use.
- Compatible with Windows, Linux, and Mac.
- General purpose and used for different domains.
- Actively maintained with a large user community. Version 4 was released in October 2019, and has since had multiple updates [49].
- Simple to set up and use.

Airflow was chosen as a textual SWFMS because it is:

- Open source and free to use.
- General purpose and can be used for multiple different domains.
- Actively maintained with a large user community. Version 2.0 was released in December 2020 [50].
- Simple to set up and use, as the workflow can be installed using pip, the package installer for Python, and created using Python code.

Airflow also supports Linux and Mac. For Windows, tools such as Windows Subsystem for Linux (WSL), Docker [51], or similar must be used to work with Airflow.

A couple of other SWMFSs were also explored in the course of this master project, including Kepler and Pegasus. However, on the computer used, MATLAB nodes in Kepler produced errors that was not easily resolved. There were also errors with the setup of Pegasus with Condor. Other management systems, such as Galaxy and Taverna are aimed towards bioinformatics and were not considered, as this master thesis is concerned with ocean acoustics. This lead to the choice of KNIME and Airflow, which were simple to set up and had comprehensive user guides and documentation. Another benefit, is that the software is regularly updated and thus likely to be useful in the future.

2.3.1 KNIME - The Konstanz Information Miner

KNIME is a free open source tool for developing scientific workflows. It is based on Eclipse and uses the plugin concept to enable the creation of extensions. It has a GUI where nodes can be dragged and dropped to a canvas. The nodes can be configured to the needs of the workflow by setting parameters, such as file path for the file reader node or configuration code for scripting nodes. Scripting nodes can contain configuration code from various languages depending on which node that is used. Examples of available scripting nodes are MATLAB, Python, and Java,

In Figure 2.8, a file reader configuration view is shown. In the configuration view, the user or developer can configure how the input data is read, delimiters, and the file location. To use the information from the read file, the data is passed from node to node through ports. All nodes have ports, and there are two types of ports: input and output ports. The nodes can be connected by connecting the output port from one node to the input port of another. These connections transfer data between the nodes and specify the dependencies.

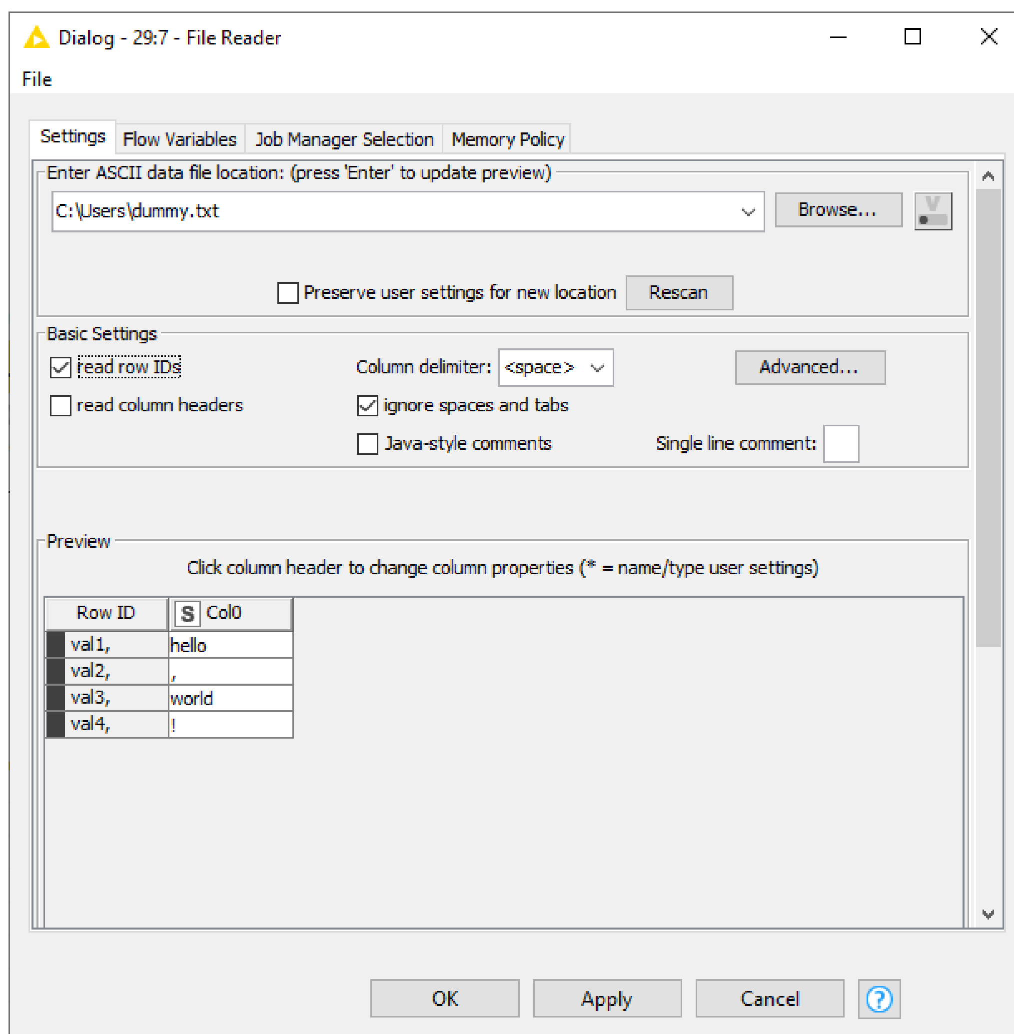


Figure 2.8: Configuration view for a file reader node in KNIME.

KNIME has numerous built-in nodes that performs different operations. In addition, there are open source KNIME extensions available that offer extra functionality, and extensions that are developed by community developers. An example of a community extension is the `KNIME MATLAB Scripting Extension` [52], that is used in the Arctic Package workflow. Some of the nodes in KNIME provide data transformation functionality, such as transposing, filtering, and splitting tables, rows, and columns. Others are input and output nodes that can read and write data, workflow-control nodes, and database nodes.

In addition to the nodes provided by KNIME, there are two types of nodes that can be created by the user: metanodes and components. Components consist of one or more nodes and can help make a hierarchical workflow model. The components aggregates steps that belong together into one descriptive node. It is also useful to reuse components for repeating parts of the workflow, as there will be less nodes to maintain. Figure 2.9 shows an example containing components from the Arctic Package workflow. Figure 2.10 shows the content within the `Run RAM model` component node from Figure 2.9. It consists of seven nodes that each represent a step in the workflow: Three MATLAB nodes, two metanodes, a MATLAB Plot node, and a node that adds the plot to a table.

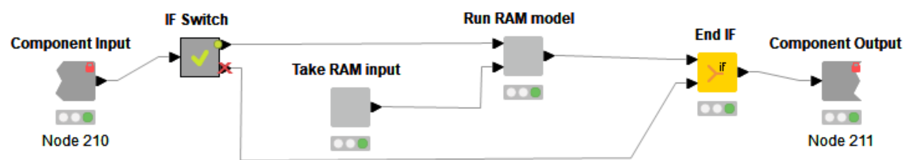


Figure 2.9: A workflow example that takes input for and runs the RAM model. The light grey nodes are components that can be expanded to see the nodes inside. The darker grey node with a checkmark is a metanode. Metanodes can also be expanded to see the nodes within.

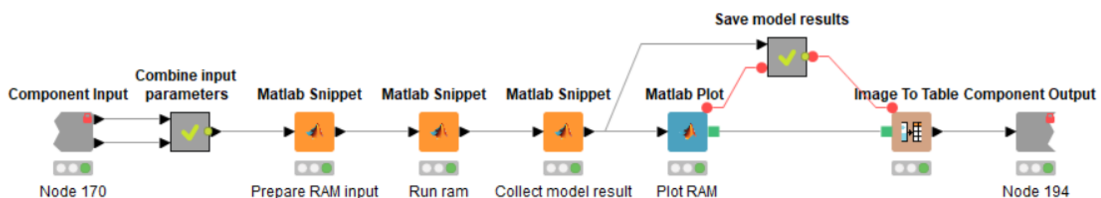


Figure 2.10: The expanded view of the `Run RAM model` node from Figure 2.9.

To create a component in KNIME, one has to select the nodes that should be included and then click `create component`. After that, it is possible to reconfigure the component and add descriptions, input ports, output ports, and change the content.

The configuration menu for components are shown in Figure 2.11. Another useful feature for components, is the `share component` feature, which makes it possible to share a component and include it anywhere in a workflow or across different workflows. If the shared component is changed, the workflows that use the shared component will fetch the update when they are loaded.

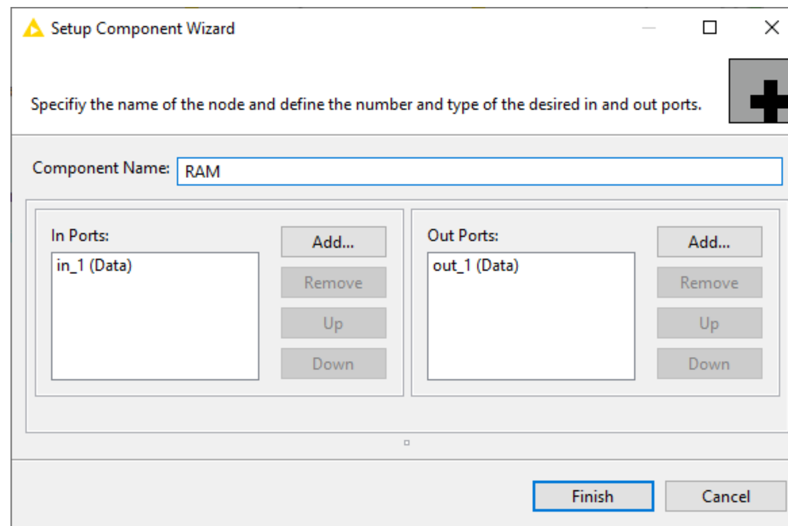


Figure 2.11: KNIME menu to set up a user created component. The top input field specifies the name of the component, shown above it in the workflow. The left box contains the user specified input ports and what kind of input it accepts. The right port contains the user specified output ports. The current input and output ports take data in the form of a table, some of the other alternatives are image input and flow variable ports.

Metanodes in KNIME are similar to components, but metanodes cannot have configuration views. Configuration views allow the component to be configured and are created by configuration nodes connected to the component. Another difference between metanodes and components, is that components can generate input and output views. Input views are graphical user interfaces where the user can set parameters for the workflow. These views can be used to generate a user interface for the workflow directly in KNIME. The interface uses the input parameters for the workflow and creates a view with a title and input field for each parameter. This is further described in Chapter 7. Output views are visual representations of the output of a node, such as images or plots.

Generally metanodes are best suited to provide an abstraction over a large workflow and they do not add any functionality, unlike components. An additional difference between metanodes and components, is that components have a clean environment without the workflow variables from the top-level workflow. Metanodes use the environment of the parent.

For running the workflow, KNIME provides the possibility to either run the entire workflow, or to run a section of it. The latter can be beneficial for debugging of the workflow or for re-running a section of the workflow with different parameters.

Another debugging advantage in KNIME is that each of the nodes stores its output and the user can easily check the output data at different points in the workflow.

With regards to control- or data-driven workflows, KNIME is a hybrid workflow that is mainly data-driven. The dependencies between the nodes are data dependencies and the workflow execution passes data form node to node. The control-driven part of KNIME is based on the possibility to add nodes that perform control functions, such as if-statements, for-loops, recursive loops, breakpoints, and case-switches. These nodes control the execution of the workflow and do not perform any data-transformations or processing. As a result, they do not fit into the data-driven domain.

Another control-driven element in KNIME, is that it is possible to force tasks to wait for other tasks, even with no data-dependencies between them. This is achieved by connecting nodes together through input and output ports and ignoring the data passed in. Another way is to use flow variables. Flow variables represent parameters in the workflow and are usually used to define user input. The flow variables are passed from node to node implicitly through the input ports. The variables can also be explicitly passed trough flow-variable ports, which exist for all KNIME nodes, excluding metanodes. If the flow variables are passed through flow-variable ports, the node receiving the flow-variables cannot run until the node passing the flow variables is finished.

KNIME is widely used in cheminformatics, bioinformatics, and image processing, and there are multiple extensions available for these domains. An example in cheminformatics is CDK [53], that builds on top of the Chemistry Development Toolkit [54] and adds functionality such as conversion to and from common formats and descriptions of chemical structures. For image processing, there is an ImageJ extension [55], which is an open-source image analysis platform. In addition to the existing extensions, it is possible to create new extensions to add new nodes the KNIME platform.

With the Arctic Package, one of the limitations of KNIME is that most of the nodes consist of one input and one output port. When working with models that require multiple different datasets as input, for example bathymetry and sound speed, handling the input and output becomes challenging. Especially when the MATLAB nodes, that were the most used node in this project, only can have one dataset as input and one as output. A workaround is to use KNIME in a more control-oriented manner, and instead of loading the data to the KNIME workflow, the MATLAB nodes load the files that are needed for the specific node. The data that is passed between the nodes in the Arctic Package workflow, is data about which files that are available and the parameters used in the workflow.

Some nodes do support multiple input and output nodes, such as the Python scripting integration, where input and output ports can be added dynamically. However, as the Arctic Package processing scripts use MATLAB, Python nodes do not solve the problem.

2.3.2 Apache Airflow

The development of Apache Airflow [44] was started in 2014 by Airbnb, and it became a part of Apache Software Foundation's incubator program in 2016. Airflow is written in Python and uses textual models to define workflows. It includes a GUI to visualize, run, and monitor workflows. The tasks in an Airflow workflow constitutes a Directed Acyclic Graph (DAG), that reflects the relationships and dependencies between the tasks. Each task is represented by a node in the DAG.

Airflow includes a metadatabase that stores metadata about the workflows and variables that can be accessed in the workflows. Additionally, Airflow includes a web server and a scheduler. The webserver interface is shown in Figure 2.12, and it can be used to monitor workflows and track task progress visually. The Airflow scheduler monitors the tasks and DAGs, and triggers tasks when their dependencies completes. As long as the database used for metadata is a MySQL 8+ or PostgreSQL 9.6+ database, multiple schedulers can be started. The default database with Airflow is SQLite, where only one scheduler can run at a time.

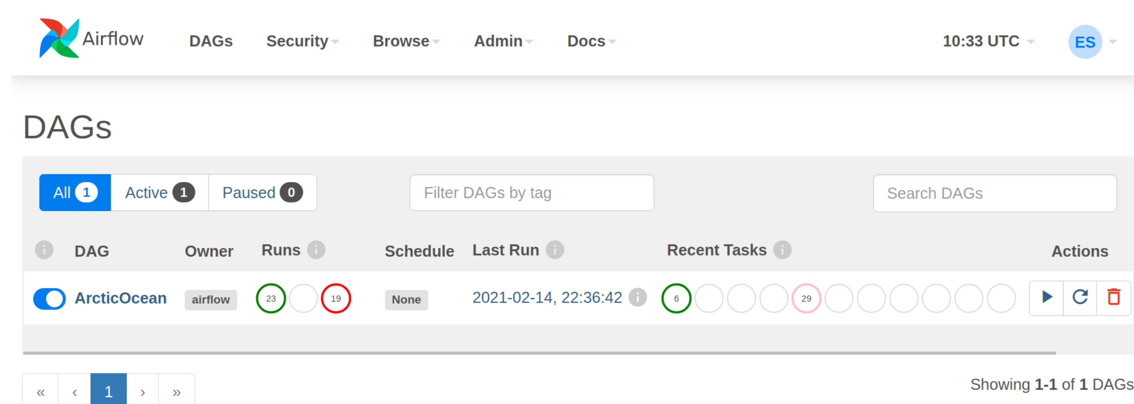


Figure 2.12: The Airflow webserver main page. Airflow contains one DAG, ArcticOcean, which has 23 successful runs and 19 failed.

Airflow is closest to the control driven domain, as the tasks do not move data between them. Instead, the connections between the nodes determine the order of execution and the relationship between the nodes. As an example, it is possible for an Airflow workflow to define that task A runs before task B by connecting node B to node A. However, the workflow can use cross-communication (XComs) to exchange messages and data between tasks by pushing data to and pulling data from the XCom database. Whenever data is pushed, it becomes available to all tasks in the workflow.

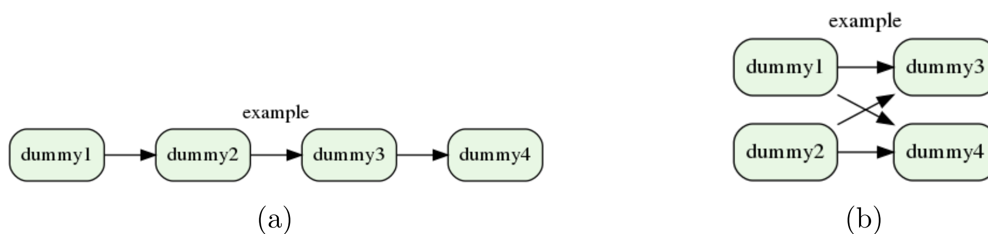
The tasks in Airflow have the same purpose as the nodes in KNIME. They perform a processing step in the workflow. In Airflow, there are different kinds of operators that can be used to perform the tasks, such as `BashOperator`, `PythonOperator`, `EmailOperator`, `SimpleHttpOperator` and different types of database operators. Additionally, it is possible to define custom operators. The operators are generally independent of each other, and the dependencies between them are determined by the underlying DAG. Airflow also supports branching, subDAGs, and task groups.

Branching is useful to select a workflow path, while subDAGs and task groups are useful for repeating workflow patterns.

To set the relationship between tasks, the composition operators `<<` and `>>` are used. The Airflow documentation refers to them as *bitshift operators*, but they are used to compose operators. Another way to set relationships between tasks is to use `set_upstream(task)` or `set_downstream(task)`, but the composition operators are recommended. The code example in Listing 2.1 illustrates how the compositions operators can be used. Lines 2-5 creates dummy operators, which are operators that do nothing. Line 7 and 8 are two equivalent methods of setting the tasks to run sequentially from task 1 to 4 as shown in Figure 2.13a. Line 10 and 11 shows two equivalent ways of setting task 1 and 2 to run before task 3 and 4 can run, as shown in Figure 2.13b.

```
1   # Create some dummy operators
2   t1 = DummyOperator(task_id='dummy1', dag=dag)
3   t2 = DummyOperator(task_id='dummy2', dag=dag)
4   t3 = DummyOperator(task_id='dummy3', dag=dag)
5   t4 = DummyOperator(task_id='dummy4', dag=dag)
6
7   t1 >> t2 >> t3 >> t4
8   chain(t1, t2, t3, t4)
9
10  [t1, t2] >> t3
11  [t1, t2] >> t4
12  cross_downstream([t1, t2], [t3, t4])
```

Listing 2.1: Python code for composing operators to decide their relationship.



(a) (b)

Figure 2.13: Example of Airflow task compositions. (a) shows the Airflow DAG where each task depend on one previous task. (b) shows the Airflow DAG where a set of task depend on another set of tasks.

To handle user input, Airflow uses variables and configurations. The variables are stored globally in Airflow and are best suited to set global settings for the workflows. The `dag.run()` method also takes an argument `conf`, which is a user defined dictionary passed when triggering the execution of a DAG. The DAG can access the configuration using Jinja templating. A Jinja template is a text file that contains

values and expressions. When the template is rendered, the variables and expressions are replaced with values. The code example in Listing 2.2 shows how Jinja templating can be used to print the DAG configuration value my_input.

```
1 input_task = BashOperator(
2     task_id='input_task',
3     bash_command="echo {{ dag_run.conf['my_input'] }}",
4     dag=dag,
5 )
```

Listing 2.2: How to create a BashOperator using Airflow syntax.

In December 2020, Airflow 2.0 was released. Version 2.0 includes a fully specified REST API, that was experimental in previous versions. The API supports Create, Read, Update, and Delete (CRUD) operations on the Airflow metadata objects, such as DAGs, DAG runs, and variables. The API can be used to do the same operations as can be done from the GUI, for instance list DAGs, trigger a DAG run, get the tasks in a DAG, update a DAG, and handle variables. The API also supports authentication methods, with the default being to deny all requests to the API. Authentication can be set to basic, allow all, or Kerberos. It is also possible to create a custom API authentication module.

Airflow 2.0 also included a new concept, task groups, which can be used to group tasks together visually. Task groups are similar to the subDAG feature, but the subDAG launches a separate DAG with its own environment, while task groups reside in the original DAG. This means that task groups are mainly used to clean up the visual representation of the DAG, while subDAGs have a wider range of possibilities. If task groups and subDAGs are compared to KNIME, task groups are similar to metanodes and subDAGs are similar to components. Task groups are also less complex than subDAGs and easier to implement, which makes them a better choice for grouping and organizing the DAG.

2.4 Related Work

Scientific workflows can be used in several domains and with different approaches. SWFMSs such as Pegasus and Airflow use textual specification, while Kepler and KNIME use graphical specification to model workflows. This section summarizes research related to the work undertaken in this master project. Section 2.4.1 describes related work in the scientific workflow domain and Section 2.4.2 describes extensions made to the Arctic Package.

2.4.1 Scientific Workflow Modeling

The KNIME SWFMS is frequently used within the cheminformatics and bioinformatics domains, for example in [56]–[60]. However, KNIME has also been used in other domains, such as in [61] where Jara et al. used KNIME for knowledge discovery from data from the SmartSantander smart city testbed, and for environmental modeling in [35].

This master project is also concerned with environmental modeling, but where we use KNIME for ocean modeling, Radosevic et al. [35] used KNIME for solar radiation modeling through Solar Analyst. Solar Analyst is an environmental model that assesses solar energy potential. In the paper, they describe how scientific workflows can be used to break down the modeling process into multiple nodes, thus increasing reproducibility and transparency. Radosevic et al. also integrates model validation into the workflow, by checking that the output matches a ground-truth dataset. Lastly, they describe how machine learning through the use of decision trees can be used to make parameter choices. The parameter choice is crucial to achieve accurate solar radiation estimations, and machine learning can assist in setting appropriate parameter values for spatial, temporal, and physical parameters. Setting the parameters would otherwise heavily rely on domain expertise, which makes it error prone for less experienced users.

In marine sciences, Kepler, another SWFMS that uses graphical workflow specifications, was used by Liu et al. [62] to organize and integrate different marine sensors and collect data into an automated context. The project parses heterogeneous data packets according to a communication protocol in order to automatically parse and send them to a database to be stored. They also developed a customized actor, `SensorDecoder`, in Java, which performs the parsing of the data from the sensors. Actors in Kepler are similar to nodes in KNIME. The project also created an adapter that made it possible to add new marine monitoring sensors to the network.

Kepler can also be used for the machine learning domain. Nguyen et al. [63] developed a Kepler workflow to integrate multiple implementations of the machine learning algorithm k-means into one workflow. This enables users to compare the performance of the different implementations of the algorithm. They used implementations in R, MLlib, Mahout, and KNIME. For the KNIME case, they built the workflow in KNIME and invoked it from Kepler with an external execution actor. The result is a workflow where the different implementations can be compared, and the best performing implementation can be selected for the specific case, e.g., for a large dataset.

For textual workflow specifications, the scientific workflow management system Pegasus was used by Brown et al. [64] to search for black holes and neutron stars. The wave data was comprised of several terabytes and there was a lot of distributed computing resources available. They used the Pegasus SWFMS to describe the workflow and a collection of Python modules, Glue, to assist in building workflows. Glue contains pipeline modules that takes analysis parameters and metadata as input and outputs Pegasus workflow files (DAX files). They also extended Glue to work in a grid context.

Apache Airflow also uses textual workflow specifications, and was used by Dosset et al. [65] to analyze data collected with Belle II, a detector that collects data from e+e- collisions made by the SuperKEKB accelerator. Calibration of the data before analysis is important to get accurate results, but prompt datasets allows scientists to begin analysis earlier and then switch to the well-calibrated datasets for final results. Doset et al. used Airflow to automate prompt calibration of payloads to Belle II to reduce human error in the calibration, and thus increase the performance for the prompt datasets. To automate the process with Airflow, they also developed a Python package *b2cal* as a custom plugin to Airflow, and extended the Airflow webserver with custom pages for interacting with the prompt processing and calibration tasks.

Onkykiy et al. [66] used Apache Airflow as a tool to organize a workflow to collect, process, and analyze large amounts of data in a scientific field. The goal of the project was to create an architecture and design for a multiagent system (MAS) for information and analytical support. The project is separated into three main parts: software agent communication, workflow planning and monitoring, and data storage. Apache Airflow is used for the workflow planning and monitoring-part of the project. The paper shows that Airflow is suited for the project.

In [34], Vigder et al. encountered similar challenges to the ones in the Arctic Package. They had a complex system with numerous scripts with little version control, and software tools implemented in Fortran or C. They developed Sweet, a SWFMS with the goal of collecting, analyzing, and managing data and creating reports. The Sweet SWFMS used textual workflow specifications written in Python for setting up workflows. The results showed that productivity increased and that they needed fewer workflows to perform the same tasks compared to earlier.

Callaghan et al. [67] did a case study of an earthquake science application for CyberShake, a probabilistic seismic hazard computational system. They identified a set of metrics to help discover areas of optimization and measure if changes improved the performance. They divided the metrics into three categories: task-level metrics, workflow-level metrics, and application-level metrics. The metrics categories gives a better understanding of which factors that contribute to which performance aspect. Using the metrics, they were able to identify bottlenecks and find possible solutions. One metric used was the number of reads and writes to file (I/O operations). They captured I/O operations and looked for high I/O demands, and when they found a bottleneck with a high demand, they successfully reduced the number of I/O operations which lead to an increase in performance for the workflow. Other metrics they used were memory usage, CPU usage, delays per job, and parallel speedup.

The literature about scientific workflows covers different domains and use cases. It also covers how scientific workflow can impact attributes of the software, such as transparency, productivity, and monitoring of workflows. This master project contributes to investigate environmental modeling through ocean acoustic models as scientific workflows, and attempts to assess the impact scientific workflows has on the software quality of a complex software package.

2.4.2 Extensions of the Arctic Package

Van den Bergh [19] and Klockmann [20] extended the Arctic Package with new acoustic models, the possibility to save results, and changed the GUI to make it more user friendly. They used the design science research methodology to gain insight about the current GUI and identify aspects that could be improved.

One of the elements they added was a `Save Result` button in the GUI and a corresponding MATLAB script, `saveResults.m`, to save results. The script stores all data and parameters the model used and saves it to a MATLAB data file with a .MAT file extension. They also cleaned up the GUI visually, by making the appearance more similar across different operating systems and extracting functionality that was rarely used to an advanced settings tab. The model specific settings were moved to a separate window, such that the parameters for the models not in use were hidden. Another addition they made was to add time steps. The propagation of the sound varies with seasonal changes, and they added the possibility to run a model for a specific time period or for an annual average.

New models (RAM, MPIRAM, and Bellhop) were added to the Arctic Package. The models were previously implemented, but Van den Bergh and Klockmann created adapters in order to fit the models into the Arctic Package. The new models provide some degree of modularity, as the new code mostly is located in the model-specific files, `runRAM.m`, `runMpiRam.m` and `runBellhop.m`.

Chapter 3

Problem Description and Analysis

The CAATEX project collects information about the Arctic, and there is a need for a system that can process the data and present the results. The data contains information about ice coverage, temperature, salt, currents, and results from the acoustic measurements, such as return time and transmission loss. The data is used to create acoustic models to better understand the propagation of the sound through the ocean. The Arctic Package contributes to fill the need for modeling of data for NERSC, but it is not as efficient and simple to use as desired.

This chapter describes the limitations of the original Arctic Package and the goals for the scientific workflow implementations of the package in this master project. Section 3.1 gives an overview of how the package is used and the functionality of the package, Section 3.2 describes the architecture of the original package, Section 3.3 introduces the data and databases used in the Arctic Package, Section 3.4 describes the existing functionality in the package and the requirements for the new version, and Section 3.5 describes how scientific workflows can be used to create a new version of the Arctic Package.

3.1 The Arctic Package

The Arctic Package was intended to simplify acoustic modeling by fetching data, executing models, and plotting the results of multiple different models without manually configuring the input files to the different models for each run. It can be used to run the acoustic models Bellhop, Eigenray, KRAKEN, RAM, and MPIRAM, as described in Section 2.1. When the Arctic Package is run, a GUI, shown in Figure 3.1, is shown. From the GUI it is possible to select the configurations needed to run the models. First, a map of the area covered by the selected database is created. The map is intended to make sure that the area to model is covered by the databases. Then, the virtual sources and receivers are plotted on the map, as well as the possible source-receiver paths. The last step is to select a propagation model to use, which in turn prompts the user to select a source and a receiver, before commencing the simulations. The Arctic Package then runs the selected model and shows a visualization of the results from the selected model.

The main functionality in the Arctic Package is shown in Figure 3.2, including the database files and parameters used in each step. The figure was developed from

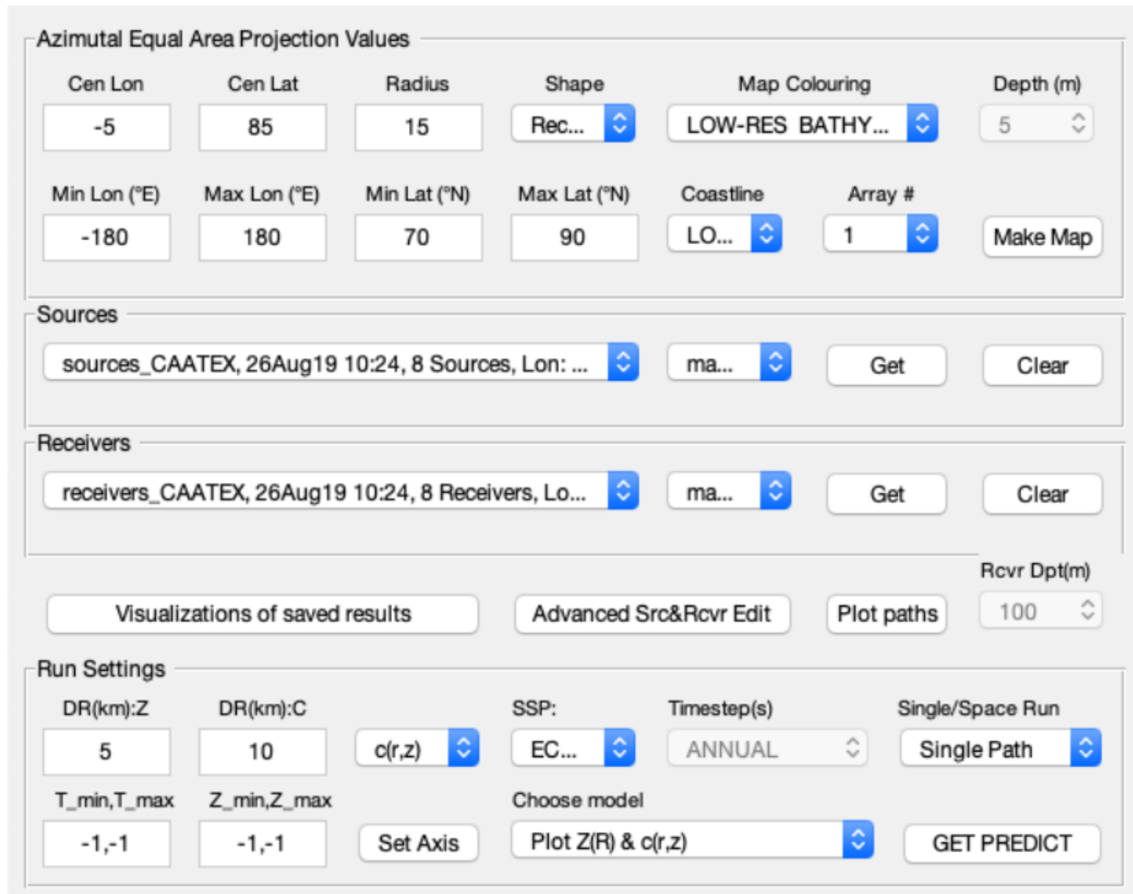


Figure 3.1: The main GUI in the MATLAB implementation of the Arctic Package [20]

descriptions of the package from Espen Storheim at NERSC and by using the GUI and source code to identify parameters and data used by and outputted by the different steps in the Arctic Package.

3.2 Software Architecture

The processing scripts of the Arctic Package are implemented in MATLAB, but the acoustic models that the Arctic Package uses are implemented in Fortran and the input and output to the models are text files. The Arctic Package provides a common setup for all the acoustic models, with the possibility to configure specific model parameters. The setup is handled through the GUI, shown in Figure 3.1.

The models used in the Arctic Package are implemented in Fortran. To compile and run Fortran code, a Fortran compiler is required. In Windows, this includes installing and setting up MinGW or Cygwin, while in Linux or macOS, the gfortran packages can be directly installed. This makes the Arctic Package harder to set up on Windows, and also requires different setups for different operating systems. The MATLAB code must also account for the different syntax when using different operating systems, as shown in Listing 3.1. Lines 1-8 start the Bellhop model if the computer runs on a Unix based operating system, while lines 9-13 starts the Bellhop model for a Windows operating system. This means that there is double the code

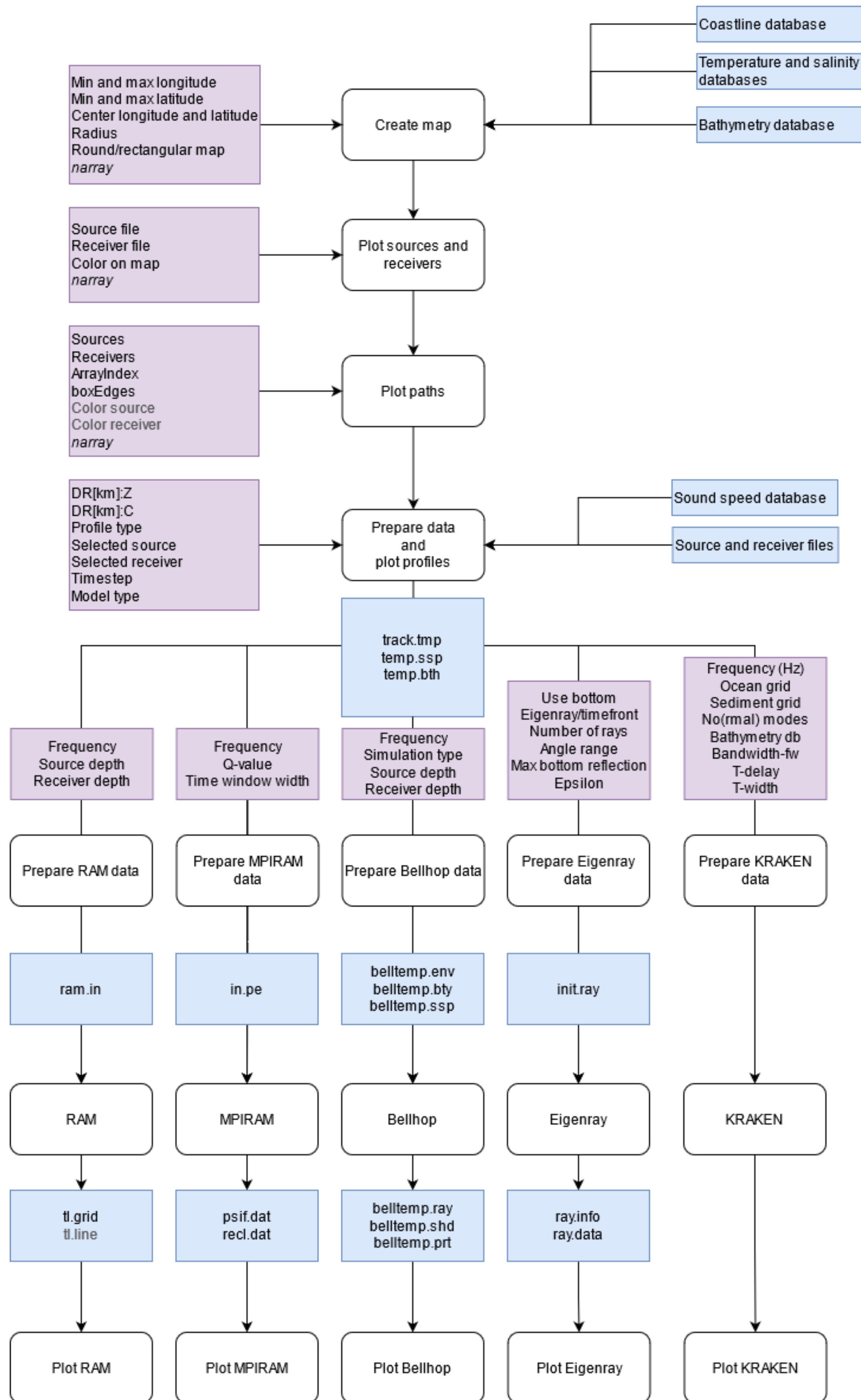


Figure 3.2: The current structure of the Arctic Package. The white boxes represents a step executed in the Arctic Package. The purple boxes represent input parameters. The blue boxes represent database files used by the given step. The text in grey in the blue and purple boxes are files or parameters that are generated, but not used by the Arctic Package.

for each model run.

```
1    if isunix,
2        if exist('belldone') == 2,
3            !rm belldone
4        end
5        ! (src/Bellhop/bellhop.exe belltemp; touch belldone ) &
6        while exist('belldone') ~= 2,
7            pause(2)
8        end
9    else
10       ! set
          ↪ LD_PRELOAD=/usr/lib64/libgfortran.so.3:/usr/lib64/libgomp.so.1
11       ! set OMP_NUM_THREADS=40
12       ! C:\cygwin64\bin\bash.exe --login -c "cd
          ↪ /cygdrive/c/Arctic.Ocean/ && src/Bellhop/bellhop.exe
          ↪ belltemp
13   end
```

Listing 3.1: How the Bellhop model is started from MATLAB. Lines 1-8 runs the model if the OS is macOS or Linux and lines 9-14 runs the model on Windows.

Table 3.1 shows a breakdown of what the package contains, measured by CLOC [68]. The Arctic Package contains a large amount of files, and is a mix of MATLAB, Fortran, C, and shell scripts. The HTML files are documentation and the CSS code belongs to the HTML documentation files. In total, the Arctic Package contains 334 source files, and an additional 147 files that were either empty, datafiles, non-HTML documentation, or PDF files. This gives a total of 508 files. In total, the 334 source files contains 28 761 source code lines, without counting blank lines and comments. Including blank lines and comments, the package contains 46 269 lines. CLOC also reported that the Arctic Package contained an Assembly file, but the file was a datafile with the extension `.S`, which CLOC recognized as Assembly. The file was excluded from the table, as the other datafiles were not counted by CLOC.

Table 3.2 shows how the Arctic Package is structured. The bulk of the Arctic Package MATLAB code is located at the root level and in the `mfiles` directory. The folders contains the files that control the GUI and the processing scripts and functions. All of these files are intertwined with the main GUI and fetches input parameters directly from it. The Fortran source code for the acoustic models is located in the `src` folder and some of the models also include MATLAB scripts for testing or processing of the results. The executables for Eigenray and KRAKEN are placed in the `bin` directory. The directories `priorRuns`, `dataBase`, and `savedResults` contain datafiles, while the `modes` directory contains additional MATLAB scripts and input data to the scripts.

Most of the files in the Arctic Package are MATLAB scripts, not functions that

Table 3.1: The analysis output of the source code created by CLOC [68] on the full Arctic Package. The analysis skipped 147 files that did not contain source code. Code lines excludes blank and comment lines, while total lines includes them.

Language	**Files**	**Code lines**	**Total lines**
MATLAB	222	15 414	24 929
HTML	5	4 071	5 776
Fortran 90	65	5 422	9 221
Fortran 77	18	2 804	4 326
Bourne shell	7	404	696
make	7	340	748
CSS	1	158	209
C	2	114	326
C/C++ Header	7	34	35
Total	**334**	**28 761**	**46 269**

accepts input and returns some output. This makes it challenging to determine which variables are created and where they are changed or used. Additionally, the scripts in the Arctic Package often launch other scripts, which makes it difficult to keep track of which scripts that act together and which are independent. As a further complication, the variables created in the scripts are stored in the workspace and available to all scripts. This makes the code hard to maintain and debug. If an error occurs, the issue may seem to be caused by a certain script, but the root cause might originate from a seemingly unrelated script. The same issue arises when a script needs to be changed. It is hard to see if the change will affect other scripts or variables they depend on, as there is no clear way to identify the dependencies of each script. Creating functions would improve this issue, as variables are local to their function. This means that all the dependencies must be passed as arguments to the function and if another script or function needs the output produced by the function, it must be returned from the function. This way it would be clear which functions use which variables and where they are changed. It would also help reduce unnecessary memory usage.

In conclusion, the current software architecture of the Arctic Package is large and complex. It is a mix of different languages, documentation, datafiles, and libraries. In addition, the package contains a number of files that are not used and files that are partly duplicated. There is also a tight coupling between the GUI and the scripts that perform calculations, and a number of scripts that work together through variables stored globally in the MATLAB workspace. All of these factors, and the lack of a setup or user guide for the Arctic Package as a whole, contribute to making the package difficult to set up and use, and even more difficult to change and maintain.

3.3 Databases Used in the Arctic Package

The Arctic Package contains a folder, `dataBase`, where the input data about the ocean and bathymetry is stored. It uses data files about the coastline, sound speed, bathymetry, temperature and salinity that originates from openly available datasets.

Table 3.2: The file structure in the Arctic Package.

Folder	Content description
root level	mkmap.m that launches the main GUI and files generated as the models are run.
src	The source code of the acoustic models in the Arctic Package. Includes their documentation and executables.
savedResults	Saved results from running acoustic models.
priorRuns	Files containing information about sources and receivers
modes	Plots of profiles for sound speed, temperature, salinity, and buoyancy.
mfiles	The files that binds the package together. Creates the map, prepares the data and runs the models. It contains subfolders runModels, plotSaved, guiParameters and extraPlots.
mfiles/runModels	Contains code to prepare input files and run models Bellhop, RAM, KRAKEN, and MPIRAM.
mfiles/plotSaved	Contains plotting information.
mfiles/guiParameters	Set up the model specific GUIs.
mfiles/extraPlots	Contains code to plot data.
mfiles/buttonFunc	Controls buttons, such as *save plot* on the figures outputted by plotting.
m_map	A mapping toolbox by Pawlowicz [69].
dataBase	Files containing data about temperature, bathymetry, salinity and sound speed.
bin	Contains an Eigenray executable and the Matlab EXecutable (MEX) file for the KRAKEN MATLAB function. A MEX file is an interface between MATLAB and functions in C, C++, or Fortran.

The datafiles are stored in either a .DAT format or a .MAT format. The .DAT file extensions indicates that it is a generic datafile to store information. The .MAT files contains MATLAB formatted data.

The bathymetry is obtained from The International Bathymetric Chart of the Arctic Ocean (IBCAO) database, described in Section 1.1.2. The Arctic Package includes three IBCAO resolutions, 30 arc seconds, 2 arc minutes, and 20 arc minutes, where 30 arc seconds is the highest resolution and 20 arc minutes the lowest. As much of the Arctic area is unknown, the package needs to approximate the bathymetry at the points where there is no data. This is done by interpolation between the known bathymetry points to the transect to be modeled. The chosen resolution determines how many points along the transect between the source and receiver that is used, and how many interpolations that are computed. By selecting the higher resolution, the computation time for running the acoustic models will be longer.

The sound speed data used in the package originates from the following databases:

the Estimating the Circulation and Climate of the Ocean version 4 (ECCOv4) [70] project, World Ocean Atlas (WOA) database [71], GECCO [72], and the Arctic Subpolar gyre sTate Estimate (ASTE) [73] dataset. The ASTE dataset is referred to as Ocean Model in the Arctic Package.

ECCOv4 use observations from satellites, instruments in water, and computer models to estimate ocean and sea ice observations. The temperature data is used to plot the ocean temperature on a map. There is also data about sound speed, salinity, and buoyancy. Buoyancy is the upwards force on an object in a fluid, which in this case is the ocean. The temperature, salinity, buoyancy, and sound speed profiles can be plotted along the bathymetry. GECCO is the German contribution to the ECCO project, and the data from 2007 was used to test the integration of the GECCO data into the Arctic Package.

WOA is maintained by the National Centers for Environmental Information (NCEI) Ocean Climate Laboratory Team (OCL). WOA provides annual, seasonal, and monthly ocean data that are used in the Arctic Package. For each of the time steps WOA provides information about salinity, temperature, dissolved oxygen, phosphate, and silicate.

The Fleet Numerical Meteorology and Oceanography Center (FNMOC) Global HYbrid Coordinate Ocean Model (HYCOM) data models the Fram-strait. The Arctic Package uses the HYCOM dataset for modeling of the Fram strait at 300 m depth by using salinity and temperature data.

The ASTE dataset is a medium resolution model of the Fram Strait. In the Arctic Package data about temperature and salinity are used to calculate the sound speed. It is also possible to draw the map colored with salinity or temperature values from ASTE. The data is annual averages.

When the package was further developed by Van den Bergh, the ASTE and GECCO datasets were added, but there were bugs related to NaN values when interpolating the sound speed profiles. The bug was not resolved. Modeling using ECCO and WOA data work as expected.

3.4 Functionality and Requirements

The original Arctic Package contains a number of features. The goal of the package is to simplify acoustic modeling and simulation, but in addition to setting up and running the acoustic models, the package also contains convenient additions to simplify processes that repeat themselves, such as running a model for different receivers or for multiple timesteps.

The existing features in the Arctic Package are as follows:

1. Plotting a map of a specified area to use for modeling. The map is colored with bathymetry, temperature, or salinity data. Figure 3.3 shows an example map

colored with bathymetry data. The bathymetry shows the depth at different points in the map.

2. Plotting sources and receivers on the map. The source and receiver locations and depths are stored in plain text files (.DAT) that are loaded into MATLAB and plotted on the map. The source and receiver data can also be specified manually in the GUI. In Figure 3.3, the sources are represented by green stars and the receivers are represented by white dots.

3. Plotting paths between sources and receivers. Straight lines are plotted between each source and receiver on the map. The lines illustrate the transect where the sound travels from the source to the receiver. The paths are the white lines between the stars and dots in Figure 3.3.

4. Preparing sound speed and bathymetry data. The acoustic models require data about sound speed and bathymetry. The data is read from database files and processed to a format readable by the models. This includes interpolating from the points around the selected transect if there is no data available for the given point. The user also has the possibility to select which bathymetry resolution and which sound speed database to use.

5. Prepare model input data. Each acoustic model requires model specific parameters, such as frequency, source coordinates, and receiver coordinates. The input data is prepared and stored in model specific input files.

6. Running the acoustic models RAM, MPIRAM, Bellhop, KRAKEN, and Eigenray. The models use their specific input files and the prepared bathymetry and sound speed data files. The models store the results in output files that the Arctic Package can use to create plots of the results.

7. Processing the result files from the model runs from RAM, MPIRAM, Bellhop, KRAKEN, and Eigenray and creating plots of the data. This reads the data from the output files from the model run and uses the outputted data to create plots of the relevant data.

8. Plotting the sound speed profiles, temperature profiles, salinity profiles, or buoyancy profiles along the bathymetry for the transect between the selected source and receiver. Figure 3.4 shows an example of such a plot.

9. Running acoustic models for multiple receivers. This feature creates N receivers distributed evenly along the circumference of a circle with a given radius r. Both of these parameters are selected by the user. The Arctic Package then runs the selected model for each of the transects between the source and the receivers. This gives N results for N evenly spaced receivers at radius r distance from the source.

10. Running acoustic models for different timesteps. This feature is only available for the sound speed databases WOA and GECCO2007. The different timestep settings are: annual, a selected month, a selected season, seasonal, or monthly. The annual option uses the annual average data, the season and month options uses the data from that season or month, the monthly option uses the data

from each month and runs the model once for each of the months, and the seasonal option runs the model once for each season. If the models are run for multiple timesteps, the result is multiple plots, one for each timestep.

11. Saving model results into a .MAT file, which is the native MATLAB data format. The file contains information about which model it is, the sound speed, bathymetry, model parameters, and the output from the model. The parameters stored depends on the model.

12. Visualize saved results. The saved .MAT file is loaded into the workspace and the user can select a plot to display for the saved results. In the current version of the Arctic Package, there are some extra plot options from saved files, such as plotting sound speed anomalies and comparison of Eigenray travel times.

13. Running two different models and plotting the results in one plot. The combinations possible are RAM together with the Eigenray eigenray option and MPIRAM together with Eigenray timefronts option. Figure 3.5b shows the RAM and Eigenray eigenray option plot, while Figure 3.5a shows the MPIRAM and Eigenray timefront option plot.

Even with the convenient additions to the Arctic Package, it is mostly used for demonstration and educational purposes, and only to a limited extent by the researchers at NERSC. One of the reasons for this is that with the typical use the package involves running different simulations for the models, such as for different sound speed databases. Running multiple simulations in the Arctic Package as it currently is, involves waiting and a lot of button clicks. The GUI needs to be configured every time a model is run, which takes time and is prone to user errors. Another issue is that the Fortran models require extra setup and code.

To improve the package, the main goal is to create a scientific workflow to improve and automate the code execution. However, there have also been some requests for new features in the Arctic Package:

- Possibility to override source and receiver depths when running acoustic models.

- Running the same model for different sound speed databases in the same model run.

- Simplify adding new models to the package.

For the workflow versions of the Arctic Package, the goal is to keep as many as possible of the existing features and then look into the new features. The most important functionality is to prepare and run acoustic models, which is prioritized. Another goal in this thesis, is to simplify the process of setting up and using the package for the user. This can be achieved by improving the documentation of the package, which in turn will help with maintenance and can simplify the process of adding new features.

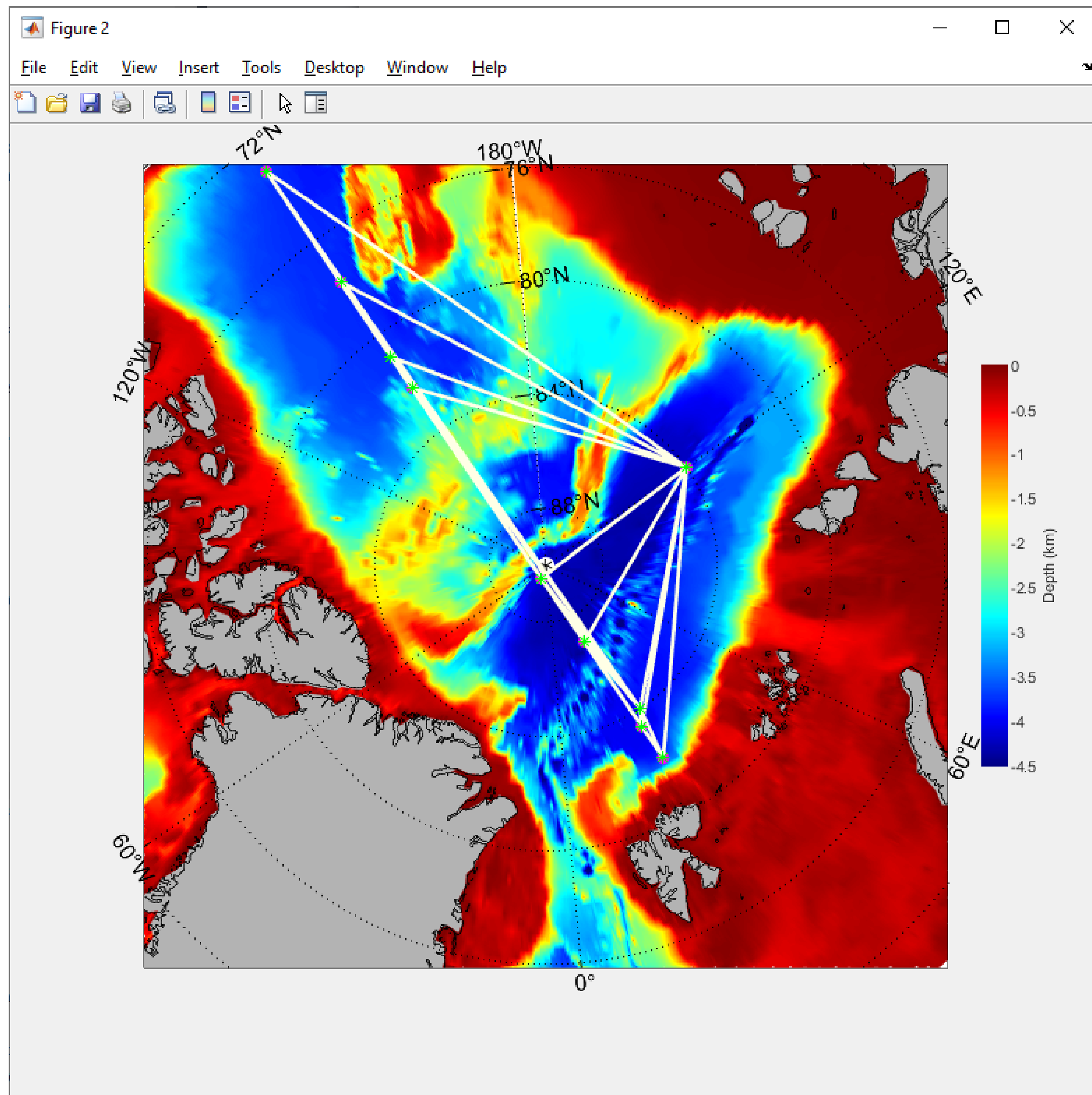


Figure 3.3: A map created with the Arctic Package. It uses the IBCAO database for the coloring of the area which shows the depth. The red areas represent the shallow regions, while blue represent the deeper regions. The green stars represent the sources, the magenta dots represent receivers, and the white lines represent the paths between the sources and the receivers. Some of the source and receiver coordinates are overlapping.

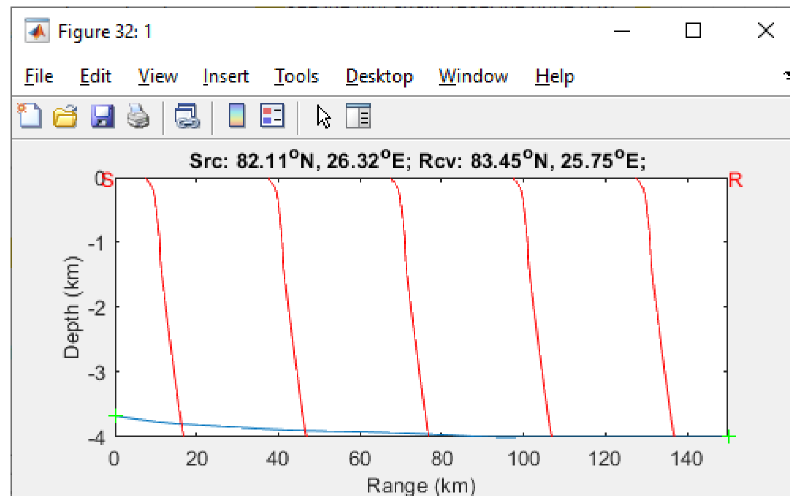
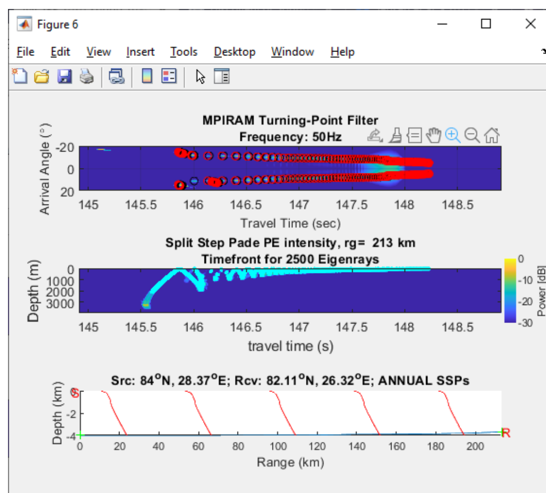
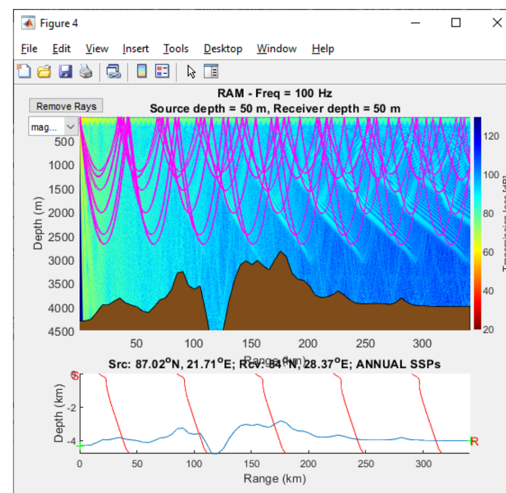


Figure 3.4: The bathymetry and sound speed profile on the transect from a source at 82.11° North and 26.32° East and a receiver at 83.45° North and 25.75° East. The sound speed profiles are the red lines, while the bathymetry is the blue line.



(a)



(b)

Figure 3.5: Plotting two model calculations on top of each other. (a) shows MPIRAM and timefronts calculated by Eigenray in the same plot. The top plot shows a turning point filter for MPIRAM in light blue and for Eigenray timefronts in red and black. The middle plot shows the timefronts for the eigenrays, calculated by Eigenray. (b) shows RAM and eigenrays calculated by Eigenray in the same plot. The transmission loss from RAM is plotted in blue and yellow, while the eigenray paths are plotted in pink.

3.5 The Arctic Package as a Scientific Workflow

Scientific workflows can contribute to make the Arctic Package more manageable. They provide a means to document each step within the acoustic modeling and computation process and to visualize the process. This makes the process easier to understand, both for programmers and non-programmers. It can also help make the code more maintainable, as the relationship between the tasks become more explicit. Another benefit that scientific workflows provide, especially the graphical approach, is that adding new functionality to the code can be more easily performed by a non-programmer, as the process consists of dragging and dropping nodes to a canvas.

To adapt the Arctic Package to a scientific workflow context, the MATLAB source code needs to be restructured and the MATLAB GUI decoupled from the processing scripts for the acoustic models. Decoupling the GUI will increase the modularity of the package and enable developers to update the GUI and processing scripts without them interfering with each other. Furthermore, the MATLAB scripts can be rewritten to functions, which will provide a clearer way to see which processes that depend on each other and which parameters they use. Another benefit of using functions, is that they only have access to the local variables and not the variables stored in the workspace. This would have the effect that the changes applied to parameters within the functions do not affect other functions or scripts. To rewrite the scripts, it is important to determine which tasks that are independent and which tasks that depend on each other. The tasks represents the steps in the finished workflow and the connections represent the dependencies between them.

To find the best way to utilize scientific workflows for the Arctic Package, two workflow models are developed, one in KNIME and one in Apache Airflow. As described in Section 2.3, KNIME was selected as a SWFMS that uses a graphical workflow specification and Apache Airflow was chosen as a SWFMS that uses a textual workflow specification. Apache Airflow uses Python syntax to create the workflows, and requires more programming knowledge from the user than KNIME, where the developer can create workflows by dragging and dropping nodes to a canvas.

Chapter 4

KNIME Workflow Model

The KNIME Analytics Platform is used to create a workflow and a corresponding GUI from the Arctic Package. This chapter explains the structure of the workflow and what the building blocks are. Section 4.1 gives an overview of the main components of the KNIME workflow, Section 4.2 explains more in detail what each of the components do, and Section 4.3 describes the changes made to the MATLAB source code in the Arctic Package.

4.1 KNIME Workflow Overview

Figure 4.1 shows the top level workflow in KNIME. The KNIME workflow covers activities 1-8 and 11 of the Arctic Package requirements from Section 3.4. Activities 5-7 includes preparing, running, processing, and plotting of model results and is implemented for four of the five models, where KRAKEN is the model not added.

When attempting to refactor the code for the KRAKEN model, issues were encountered. Unlike the other models that run the Fortran code directly from MATLAB, the KRAKEN model uses a MEX file to call the KRAKEN Fortran code, and a C routine as the entry point to the Fortran implementation. The different structure meant that there would have to be created new MATLAB functions for generating input files for the model and to process the output. Originally, KRAKEN has a lot of configurable parameters, but in the Arctic Package only a few were available to configure and the rest were hard-coded to specific values. Additionally, the new input generation scripts turned out to be unstable. These challenges contributed to the KRAKEN model not being prioritized for the workflow version of the Arctic Package. Additionally, the process of adding the KRAKEN model to the workflow is largely the same as demonstrated for the other models and is likely to work if more time is spent developing the processing scripts for KRAKEN.

The top level workflow represent the main tasks in the workflow and abstracts away the details of the workflow implementation. The components and metanodes can be expanded to see the steps within. There are also if-statements within each model component to control the data-flow and which models that are run. The top level workflow is divided into eight main components:

- The `Take workflow input` node, which generates a GUI with the workflow parameters that can be set by the user.

- The `Prepare sound speed and bathymetry files` node, that prepares the input data used for all the models such as sound speed, bathymetry, and ranges and stores them to the local file system. The results are stored in text files and loaded by the MATLAB nodes when needed.
- The `Plot map` node, which plots a map of the selected region with the selected sources and receivers.
- The `Plot profiles and bathymetry` node, which plots the selected profile (sound speed, salinity, temperature, or buoyancy) alongside the bathymetry for the transect between the source and receiver.
- The `RAM` node, which creates input files for the RAM model, runs the model, and plots the result.
- The `MPIRAM` node, which creates input files for the MPIRAM model, runs the model, and plots the result.
- The `Bellhop` node, which creates input files for the Bellhop model, runs the model, and plots the result.
- The `Eigenray` node, which creates input files for the Eigenray model, runs the model, and plots the result.

In addition, the workflow collects the output from the models and stores the plots in the `Image Viewer` node. To see the output plots from the workflow model, one can right click on the `Image Viewer` node and select `View: Image Viewer`.

4.2 Workflow Components

The KNIME workflow is built up of nodes. The GUI is created using widget nodes that are collected into input components, while the rest of the workflow is created using non-widget nodes. Many of the nodes are grouped together into metanodes or components to structure the workflow.

4.2.1 Input Components

The input component consists of five sub-components, shown in Figure 4.2. The sub-components divide the input into related categories. Their main purpose is to add structure to the input process, as each of the components consist of multiple widget nodes. For example, the map input component shown in Figure 4.3 has eight input widgets, which takes up a lot of space. The other sub-components also contain a varying number of widget nodes.

In Figure 4.2 and Figure 4.3, there is an isolated node at the left side of the figures called `Component input`. The node takes the input passed from the top level workflow, and passes it to the components. However, in this case there is no input from the workflow to be passed to the widgets, as they are the ones that output the data to the workflow. This causes the `Component Input` to float by itself on the left side. The same would be the case for the `Component Output` node if there was no output

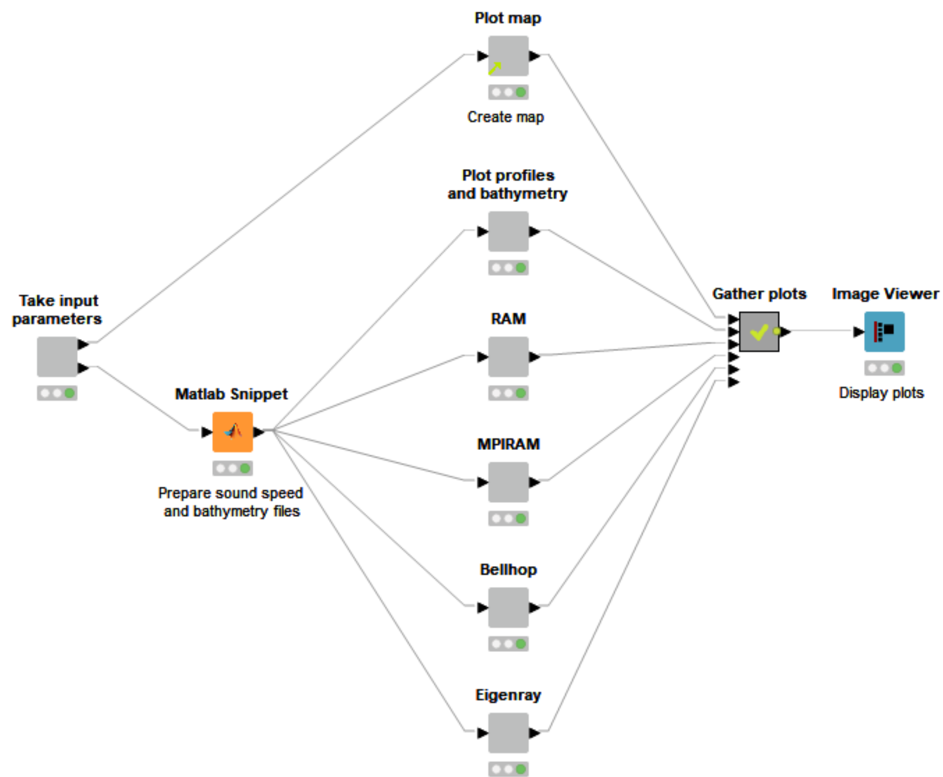


Figure 4.1: The top level KNIME workflow that prepares model data, runs acoustic models, and plots modeling results.

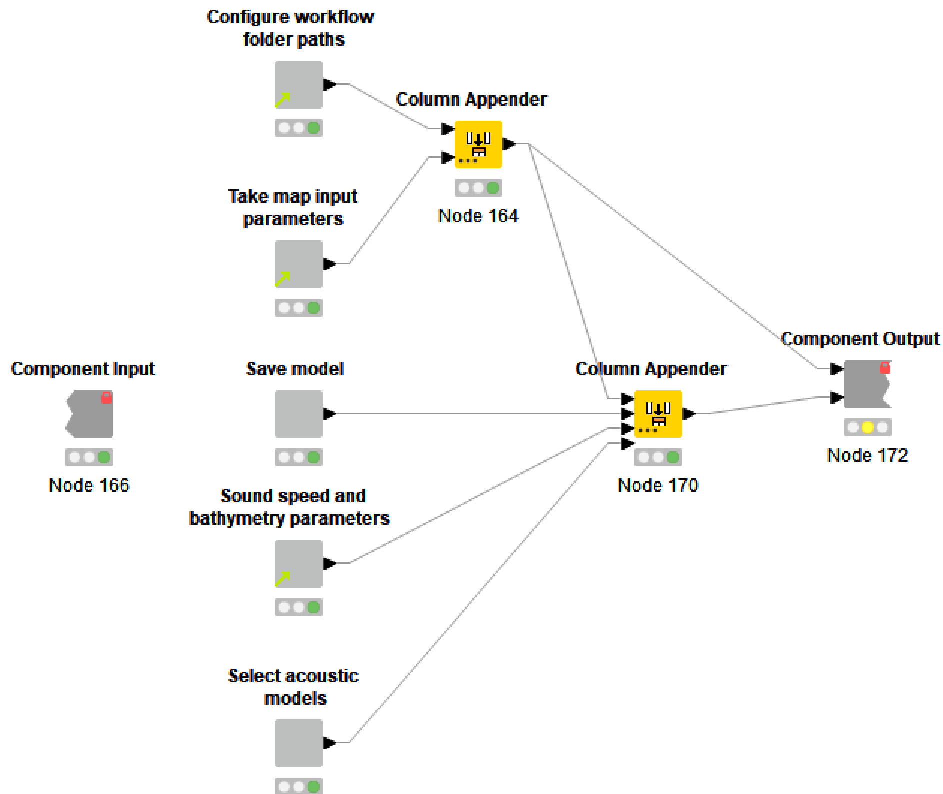


Figure 4.2: The expanded view of the `Take input parameters` node in Figure 4.1. It has five sub-components that structure the input widgets into nodes that group the input parameters together based on which parameters belong to which tasks in the workflow. The component has two output ports, as the plotting of the map and the modeling uses different input parameters.

from a component.

Widget nodes are used to take input from the user. In the Arctic Package workflow, the multiple selection widget, integer widget, double widget, single selection widget, boolean widget, and string widget are used. Each widget is configured with a label, description, and a variable name. The label and description are used as a title for the parameter in the generated GUI, while the variable name is used to access the parameter in the workflow. Default values can be set for each widget, and minimum and maximum values can be set for widgets with number values.

Each widget creates a workflow variable containing the user input. When all the user input has been collected, the workflow variables are merged together by the `Merge Variables` node in Figure 4.3, and converted to a table row by the `Variable to Table Row` node. The table row is passed between the components input and output ports as a data dependency in the workflow. The user input could be implicitly passed between nodes as workflow variables, however in this instance, the user input is passed through the input/output ports, as it is easier to access input from input ports than from workflow variables in the MATLAB Snippet nodes. Furthermore,

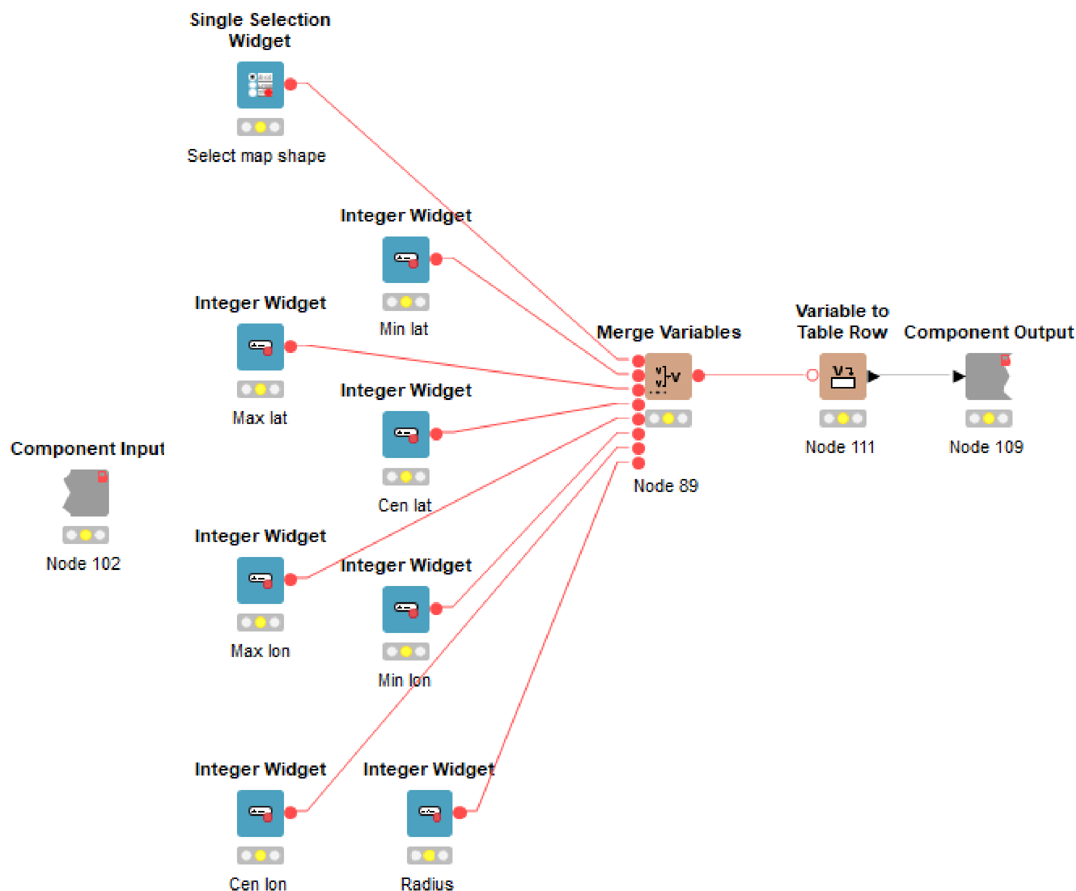


Figure 4.3: The expanded view of the `Take map input parameters` node in Figure 4.2. The node consists of widgets to take user input. The integer widget takes integer input and single selection widget creates a drop-down menu for the user.

the datasets cannot be passed from node to node through in- and output ports in the workflow, as there are not enough in- and output ports in the MATLAB Snippet node. Instead, the datasets are loaded in the nodes that use them.

The generation of the user interface for the KNIME workflow is further described in Section 7.2.

4.2.2 Map and Plot Profiles and Bathymetry Components

The expanded map component is shown in Figure 4.8. It contains a MATLAB node that plots the map and an `Image To Table` node that adds the map plot to a table. The `Plot profiles and bathymetry` component follows the same pattern as the map component, but instead of plotting the map, it plots the selected profiles and bathymetry. The `Plot map` node in Figure 4.8 plots the map with the specified coloring, sources, receivers, and the paths between them. To connect the node to the underlying MATLAB code, the configuration code view is utilized.

Listing 4.1 shows the configuration code for the `Plot map` node in Figure 4.8. The node calls the `plot_map`, `plot_rigs`, and `plot_paths` functions from the Arctic

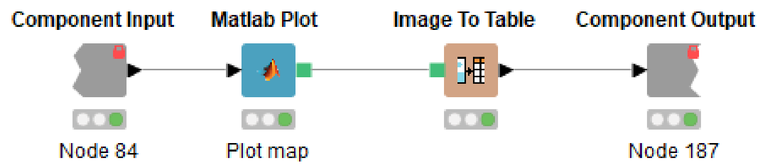


Figure 4.4: The expanded map component shows the two steps needed to output a plot of the map. The first node is a MATLAB Snippet node and creates the map and the next node adds the image to a table.

Package. In order for MATLAB to recognize the functions, the MATLAB node adds the required folders to the MATLAB path (lines 1-5). For the `Plot map` node, the files used are the functions located in the MATLAB source code directory and the data about the map located database directory. Then the node plots the map by calling the `plot_map` function (line 15), the sources by calling the `plot_rigs` function (line 16), the receivers by calling the `plot_rigs` function (line 17), and the paths between sources and receivers by calling the `plot_paths` function (line 18). Lines 21-25 add a legend to the plot, showing which colors and symbols are sources and which are receivers. The figure of the map is then saved as a PNG image (line 28).

Initially, the plotting of the map was split into six nodes, where four were MATLAB Snippet nodes. One plotted the base map, one plotted sources, one plotted receivers, and one plotted the paths. This structure was abandoned, as was inefficient. It required that the figures were saved between each MATLAB Snippet node, which added around 10 seconds to the execution time for each node.

4.2.3 Model Components

All the model components have the same structure as the RAM model component in Figure 4.5. They contain an if-switch that checks if the user specified that the model should run, an input component that takes model specific input, the component that runs the model, and an end-if node.

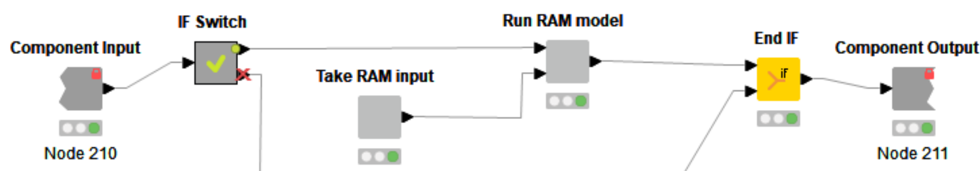


Figure 4.5: The content of the `RAM` node in Figure 4.1

Figure 4.6 shows the expanded view of the `Run RAM model` node from Figure 4.5. The steps are to prepare the model input files, run the model, process the model

```matlab
% Add the matlab files to path
matlablib = strcat(kIn.knimeworkspace, kIn.matlablib);
databasedir = strcat(kIn.knimeworkspace, kIn.databasedir);
addpath(genpath(matlablib));
addpath(databasedir);

depth = 0;
if(contains(kIn.coloring_database, 'ECCO'))
        depth = str2num(kIn.depth_ecco);
elseif(contains(kIn.coloring_database, 'ocean'))
        depth = str2num(kIn.depth_ocean);
end

fig = figure('Name', 'Map', 'visible', 'off');
plot_map(kIn.coastline_resolution, kIn.coloring_database,
    kIn.minlon, kIn.maxlon, kIn.minlat, kIn.maxlat, kIn.cenlon,
    kIn.cenlat, kIn.radius, kIn.shape, depth);
plot_rigs(load(kIn.sources), 'w', '*');
plot_rigs(load(kIn.receivers), 'm', 'o');
plot_paths(load(kIn.sources), load(kIn.receivers), [kIn.minlon,
    kIn.maxlon, kIn.minlat, kIn.maxlat]);

% Add legend
h = zeros(2, 1);
h(1) = plot(NaN,NaN,'om', 'MarkerFaceColor', 'm');
h(2) = plot(NaN,NaN,'*w');
lgd = legend(h, 'receivers', 'sources', 'TextColor', 'white');
set(lgd,'color','black');

map_png = strcat(matlablib, '/map.png');
saveas(gcf, map_png);
```

Listing 4.1: The MATLAB code in the `Plot map` node.

results, and plot the model results. The models also have the option of saving the model results to a .MAT file. The models RAM, MPIRAM, and Eigenray follow the pattern in Figure 4.6 and contain the same nodes. The difference is the naming and the configuration that connects the nodes to the underlying MATLAB code. The configuration code needs to be adapted to each model, as there are different scripts to run for each model.

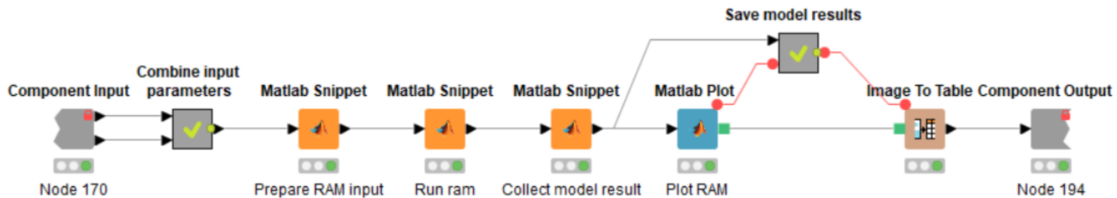


Figure 4.6: The expanded `Run RAM model` node from Figure 4.5 shows the steps needed to prepare data, run the RAM model, and plot the results.

Listing 4.2 shows the configuration code executed when the `Prepare RAM input` node in Figure 4.6 is run. It prepares the MATLAB workspace by adding necessary files and folders to path in lines 2-7, where lines 2-4 finds the absolute path to the Arctic Ocean source code folder, the database directory, and the acoustic model directory. The paths are configured to be relative to the KNIME workspace location. The files created for each model run, such as `temp.ssp` and `temp.bth` are stored at the same level as the database and added to path in line 6. Line 9 and 10 loads the selected source and receiver information, while line 12 calls the MATLAB function that prepares the RAM input file from the user input.

```
1   mOut = kIn;
2   matlablib = strcat(kIn.knimeworkspace, kIn.matlablib);
3   databasedir = strcat(kIn.knimeworkspace, kIn.databasedir);
4   modeldir = strcat(kIn.knimeworkspace, kIn.acoustic_models_dir);
5
6   addpath(strcat(databasedir, '/../'));
7   addpath(genpath(matlablib));
8
9   src = load(kIn.sources); src = src(kIn.source_idx, :);
10  rcvr = load(kIn.receivers); rcvr = rcvr(kIn.receiver_idx, :);
11
12  prepare_ram(kIn.frequency, src(3), rcvr(3), strcat(modeldir,
     ↪ '/RAM/data/ram.in'));
```

Listing 4.2: The MATLAB code for the `Prepare RAM input` node from Figure 4.6.

The `Run RAM` node starts a Bash script that launches the RAM model in a Docker container. Listing 4.3 shows the `Run RAM` node configuration code, and Listing 4.4

shows the code for the launch script. The `Run RAM` node first navigates to the RAM folder (line 3) and then it starts the Bash script `launch_ram.sh` (line 4). Lines 3-6 in the Bash script in Listing 4.4 checks if there exists a RAM image, and builds one if not. Lines 8-11 checks if there exists a container for RAM and if not, one is created. Line 13 starts the container, which runs the RAM acoustic model. The `-a` argument attaches the terminal's standard input, output, and error to the running container.

A node called `External Tool` was considered for running the launch script, but it required the path to Bash, the path to the directory to execute in, an input file, and an output file. The input and output files were not used by the Bash script, which lead to the conclusion that it was simpler to start the script from MATLAB. In addition, the paths needed to be specified even though they were not used, which meant that they would have to be reconfigured each time the package is set up on a new computer, as the paths would be different.

```
1   mOut = kIn;
2   ram_dir = strcat(kIn.knimeworkspace, kIn.acoustic_models_dir,
    ↪  '/RAM');
3   cd(ram_dir);
4   ! launch_ram.sh
```

Listing 4.3: The `Run RAM` node in Figure 4.6 configuration window.

```
1   #!/bin/bash
2   images=$(docker images -q ram)
3   if [[ ! -n "$images" ]]; then
4   # There is no RAM image, so build one
5       docker build -t ram .
6   fi
7   containers=$(docker ps --all | grep -w ram_container)
8   if [[ ! -n "$containers" ]]; then
9   # There already is no container for RAM, create one.
10      docker create --name ram_container --mount
        ↪  type=bind,source=/$(pwd)/data,target=/ram/data ram
11  fi
12  echo "Running RAM..."
13  docker start ram_container -a
14  echo "Docker finished running RAM!"
```

Listing 4.4: The Bash script launch_ram.sh that builds the Docker image for the RAM acoustic model if there is no RAM image, and runs the RAM model.

Listing 4.5 shows how the results from the RAM model in Figure 4.6 is processed. Lines 1-4 prepares the workspace. Lines 6-9 loads the source, receiver, range, and

depth, that are used by the plotting function. Line 12 reads the model results from the file `tl.grid`, which is where the Fortran model outputs the RAM model results. Lines 14-20 finds the filename, either user specified or generated, and saves the processed results to a file with that name. Line 22 replaces the filename in the input table with the filename the file was saved to, and line 23 outputs the content of `kIn` from the node.

```
1   matlablib = strcat(kIn.knimeworkspace, kIn.matlablib);
2   databasedir = strcat(kIn.knimeworkspace, kIn.databasedir);
3   modeldir = strcat(kIn.knimeworkspace, kIn.acoustic_models_dir);
4   addpath(genpath(matlablib));
5
6   src = load(kIn.sources); src = src(kIn.source_idx, :);
7   rcvr = load(kIn.receivers); rcvr = rcvr(kIn.receiver_idx, :);
8   range = load('rangeBath.txt');
9   depth = load('depth.txt');
10
11  % Load RAM results
12  res = read_ram_result(kIn.frequency, src(3), rcvr(3), range,
    ↪ depth, strcat(modeldir, '/RAM/data/tl.grid'));
13
14  filename = kIn.save_filename;
15  if isnan(kIn.save_filename)
16          filename = strcat(databasedir, "\..\savedModels\ram-",
            ↪ datestr(now, 'dd-mm-yyyy_HHMM'), ".mat");
17  else
18          filename = strcat(kIn.save_filename, datestr(now,
            ↪ 'dd-mm-yyyy_HHMM'), ".mat")
19  end
20  save(filename, 'res');
21
22  kIn.save_filename = filename;
23  mOut = kIn;
```

Listing 4.5: The configuration code of the `Collect model result` node from Figure 4.6.

Listing 4.6 shows the `Plot RAM` configuration code that plots the RAM results. Lines 1-3 prepare the workspace, line 6 loads the processed RAM model output, and line 9 plots the RAM result. The figure plot is outputted by the MATLAB plot node, and `mOut` does not need to be specified.

The Bellhop model shown in Figure 4.7 has a slightly different setup than the other models, as it has the option to run five different modeling options. To be able to run multiple model option in a workflow run, the Bellhop model wraps the modeling steps in a for-loop. The for-loop loops over each of the selected model types, collects

```
1     matlablib = strcat(kIn.knimeworkspace, kIn.matlablib);
2     addpath(strcat(matlablib, '/global'));
3     addpath(strcat(matlablib, '/models/RAM'));
4
5     % Load RAM results
6     load(kIn.save_filename, 'res');
7
8     % Plot RAM results
9     plot_ram(res.freq, res.zs, res.zr, res.r, res.z, res.ttRAM,
       ↪  res.rdRAM, res.rrRAM);
```

Listing 4.6: The `Plot RAM` node in Figure 4.6 configuration window.

the plots from the iterations, and passes the plots to the output port. In Figure 4.7, the for-loop nodes are the cyan colored nodes. The for-loop is started by the `Column List Loop Start` node and ended by the `Loop End` node. The Bellhop model also has a different saving process from the other models, because processing and plotting the output happens in the same MATLAB script as in the original Arctic Package. The configuration code in the nodes follow the same pattern as the RAM example.

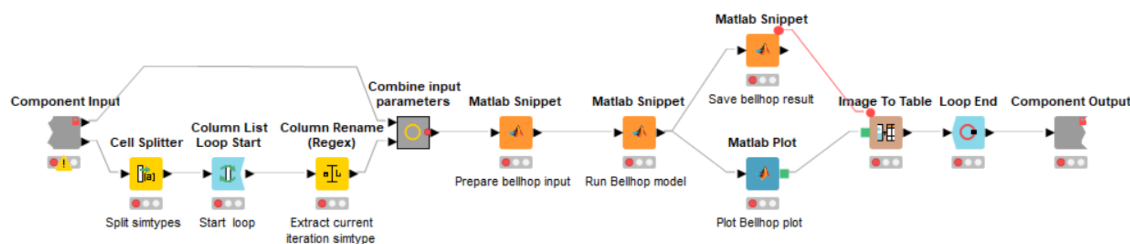


Figure 4.7: The expanded Bellhop component shows the steps needed to prepare data, run the Bellhop model, and plot the results.

4.3 MATLAB Adaptations for the KNIME workflow

Section 3.2 describes how the original Arctic Package was structured. The structure had to be changed in order to adapt the package into a scientific workflow. The file structure in the Arctic Ocean MATLAB repository in the workflow version of Arctic Package is described by Table 4.1, while Table 4.2 shows a breakdown of its contents. The original structure and complexity was described by Table 3.2 and Table 3.1 in Section 3.2.

In the original version of the Arctic Package everything was included in the MATLAB project. In the workflow version, it is divided into four parts:

Table 4.1: The file structure of the MATLAB source code in the workflow version of the Arctic Package.

Folder	Content description
models	The preprocessing of data, processing of model results, and plotting of the model runs.
map	The files used to plot the map.
libraries	The `m_map` library used by the package.
global	The files used throughout the package that are not specific to any one step in the package.
prepareInput	The files used to create the sound speed and bathymetry files the models need to run.

- Arctic Ocean - the MATLAB source code
- Acoustic Models - the acoustic model code with Dockerfiles and launch scripts for building and running the models in Docker containers.
- Workflow project - the workflow files and source code needed to run the Arctic Package workflow in the chosen SWFMS.
- Data files, source, receiver, and database files that provides information about sound speed, bathymetry, salinity, temperature, source locations, and receiver location.

Previously, all four parts of the Arctic Package were located in the Arctic Ocean project folder. As the Arctic Package is separated into four repositories, there is no need to include the folders `src`, `dataBase`, and `bin` in the Arctic Ocean repository. Additionally, the `mfiles` subfolders `runModels`, `guiParameters`, and `buttonFunc` are removed, as KNIME both runs the acoustic models and generates a GUI for the workflow. This greatly reduces complexity, by reducing the amount of files and the number of different programming languages in the Arctic Ocean project folder, as shown by Table 4.2. This is especially evident when comparing Table 4.2 to the original complexity shown by Table 3.1.

Table 4.2: An analysis of the Arctic Ocean source code created by CLOC [68]. Code lines excludes blank and comment lines, while total lines includes them. The HTML and CSS files belong to the library `m_map`.

Language	Files	Code lines	Total lines
MATLAB	125	9 043	15 879
HTML	5	2 584	3 240
CSS	1	158	209
Total	**128**	**11 785**	**16 287**

The structure of Arctic Ocean, as shown in Table 4.1, contains five new folders: `models`, `map`, `libraries`, `global`, and `prepareInput`. The `models` folder contains four subfolders: `RAM`, `MPIRAM`, `Eigenray`, and `Bellhop`. Each of the folders contains the MATLAB scripts that prepares the input files for the model, the scripts that

reads and processes the model results, and the script that plots the model results. In the original package the RAM, MPIRAM, and Bellhop files were located in the `runModels` folder, and the Eigenray and KRAKEN files were located in the `mfiles` folder. The `prepareInput` and `map` folders group files from the `mfiles` folder by function. The `map` folder contains all files related to creating the map, while the `prepareInput` folder contains the files related to preparation of the files all the models use.

In addition to changing the structure of the MATLAB source code, the original MATLAB scripts were rewritten to functions. This was necessary in order to make the connection between the KNIME SWFMS and MATLAB processing code possible. Most of the files in the original package were MATLAB scripts that used parameters stored in the workspace. When they were rewritten, information about input/output parameters and what the script does was included to improve documentation.

An example of scripts that were rewritten is the scripts `plotBath.m` and `plotBath2.m`. The `plotBath.m` script loads databases and creates a colored map, while the `plotBath2.m` plots the earth, coastline, colorbar, sources, and receivers. The two scripts were rewritten to the functions `plot_map.m`, `load_bathymetry.m`, `load_ocean_data.m`, `load_hycom.m`, `load_coastline.m`, `load_ecco.m`, and `mask_map.m`.

When the scripts were rewritten to functions, a benefit is that the function specifies which arguments are needed to perform an operation and the documentation provides information about the input and output parameters. This makes it easy to understand what the parameters are. For example for plotting a map, the function call could look like this:

```
1 function fig = plot_map(coast_resolution, coloring_database, minlon,
  ↪  maxlon, minlat, maxlat, cenlon, cenlat, radius, shape, depth)
```

This means that to call the `plot_map` function from a MATLAB node in KNIME, the parameters are known and can be inputted directly from KNIME:

```
1 fig = plot_map(kIn.coastline_resolution, kIn.coloring_database,
  ↪  kIn.minlon, kIn.maxlon, kIn.minlat, kIn.maxlat, kIn.cenlon,
  ↪  kIn.cenlat, kIn.radius, kIn.shape, kIn.depth);
```

For the original `plotBath.m` and `plotBath2.m` scripts, there is no information about the parameters used in the files or what they represent. This means that to use the script, the parameters must first be identified and their exact names must be found. To call the scripts, only one word is needed `plotBath`, but calling the script from a MATLAB node would fail, as the `plotBath` script tries to use variables from the workspace. To successfully call the `plotBath.m` script, the user input variables must be renamed and loaded into the workspace in the node. This process can be challenging, as the scripts do not specify which variables they depend on.

Rewriting the two scripts to functions made the code more self-explanatory and the naming of the functions better reflects the processing performed in them. Figure 4.8 shows the KNIME structure of a component that plots the map. The four MATLAB snippet nodes were later merged to one to increase performance, but the example still illustrates the refactoring benefit. Additionally, other complex scripts were split in the same manner.

In Figure 4.8, each node makes a call to a script. The `Plot map` node calls the `plot_map` function, the `Plot source` and `Plot receiver` nodes call the `plot_rigs` function, and the `Plot paths` node calls the `plot_paths` function. If the original scripts were used, it would only be possible to have one node even though they are two separate scripts. The reason is that the `plotBath.m` script calls the `plotBath2.m` script, and in practice there is no separation between the two tasks. The `plotBath2.m` script uses the parameters initialized in `plotBath.m`, and cannot run successfully without running `plotBath.m` first.

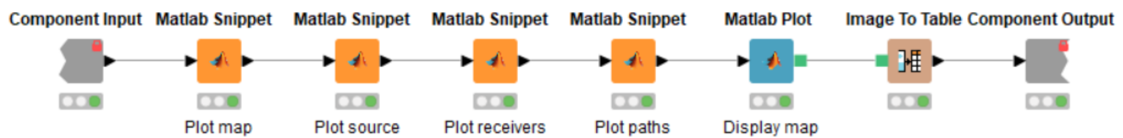


Figure 4.8: The expanded view of the KNIME map component which plots the map base, the sources, the receivers, and the paths between the sources and receivers.

Chapter 5

Airflow Workflow Model

The Airflow workflow consists of two parts, the Airflow workflow specification and a GUI implemented using the React-JSONSchema-Form framework. This chapter explains how the workflow is defined and what the building blocks are. Section 5.1 gives an overview of the main structure of the workflow, Section 5.2 explains in more depth the tasks and operators, Section 5.3 describes the REST API included with Airflow and used for connecting the workflow and the GUI, and Section 5.4 describes the changes made to the MATLAB source code of the Arctic Package to adapt it to Airflow.

5.1 Airflow Workflow Overview

The Airflow workflow covers activities 1-8 and 11 of the Arctic Package requirements from Section 3.4. It consists of two parts, the GUI generated using React-JSONSchema-Form (RJSF) and the Airflow workflow specification in Python. The two parts communicate through Airflow's REST API. To trigger a DAG run, the GUI collects the user input for the workflow and sends a request to the Airflow REST API that triggers a DAG run with the user input. This process is further described in Chapter 7.

Figure 5.1 shows a screenshot of the workflow from the Airflow webserver GUI that shows the workflow DAG. The workflow branches out to tasks that do not depend on each other, which means that the tasks in different branches can run concurrently. The tasks within a branch depend on each other and are run in the order specified by the branch. An example is that the `create_map` task and `prepare_input` task do not depend on each other, but both depend on the `get_config` node. The figure also shows that the model branches are independent of each other, but that they all depend on the `prepare_input` task.

The Airflow workflow contains two branching nodes, the `select_models` node and the `eigenray.select_plot` node within the Eigenray task group. The rest of the nodes are `BashOperators`. Most of the Bash operators use the MATLAB batch command to run the MATLAB functions needed in each task. The others use the Bash operator without MATLAB to perform tasks, such as running the Fortran models.

Figure 5.1 also shows that there are five different branches for Bellhop. This is because Bellhop contains five different model options that can be run. This means that there is a total of eight model branches, including the RAM, MPIRAM, and Eigenray branches. The Eigenray task group also has two different plot options, depending on whether the Eigenray or Timefront model option was selected. This creates a split in the Eigenray model branch as well.

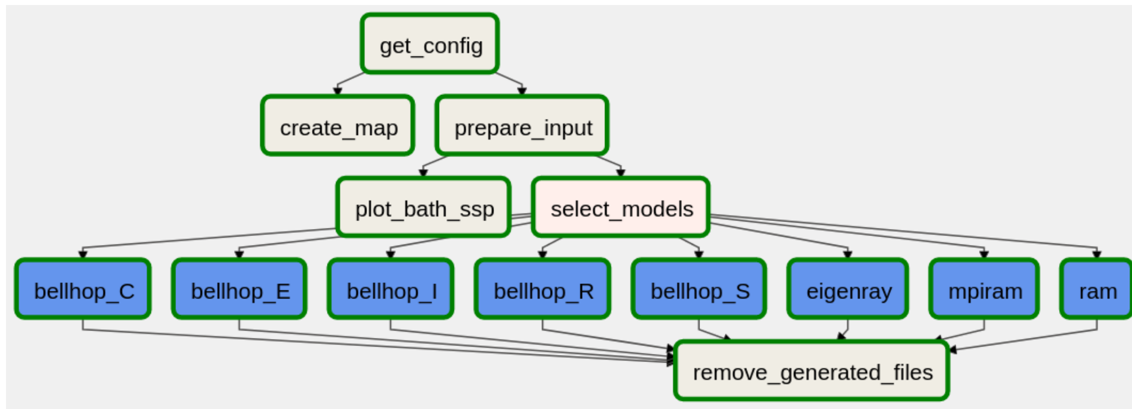


Figure 5.1: The Airflow workflow that prepares model data, runs acoustic models, and plots modeling results. The `BranchPythonOperators` have a pink coloring, and the blue nodes are task groups. Task groups can be expanded to see the tasks within. The rest of the nodes are `BashOperators`. The green border indicates that the task was executed successfully. A red border would indicate a failed task and a pink border would indicate that a task was skipped.

5.2 Workflow Operators

The operators used in the workflow are `BashOperators` and `BranchPythonOperators`. In addition, the `TaskGroup` UI grouping concept is used. Because there is no MATLAB operator in Airflow, the Bash operator is used to run the MATLAB functions from the Arctic Package.

5.2.1 BashOperator

A Bash operator executes commands in a Bash shell. In Listing 5.1, the Bash operator code for the `prepare_ram` task is shown. The Bash operators that executes the MATLAB functions follow a pattern similar to that in the MATLAB nodes in KNIME. They do the same setup by adding the project to the path before loading data- and/or source and receiver files and running the required functions. The syntax in line 3: `{{ var.json.ap_cfg.project_dir }}`, replaces the section with the corresponding configuration variable value using Jinja templating. There are also default variables that are accessible, such as `ts_nodash`, which is replaced with the execution date in ISO format.

The Bash operator takes a Bash command (line 3), which is the command run by the Airflow task. For the `prepare_ram` task, the Bash operator uses the command `matlab -noFigureWindows -sd <folder> -batch "<matlab command>"` to

run the MATLAB function. The `-noFigureWindows` argument suppresses all figures that otherwise would have popped up, and the `-sd` argument sets the MATLAB folder to the project folder specified in the `ap_cfg` variable set by the user. Line 4 adds the Arctic Package project folder to path, which ensures that MATLAB finds the function and files used by MATLAB in lines 5-9 of the code example. First, the source and receivers files are loaded and the specified receiver is selected (lines 5-8), then the Bash operator runs the MATLAB script `prepare_ram` that creates the input file for the RAM model, `ram.in`, specified as the last argument to the script (line 9).

```
1     prepare_ram = BashOperator(
2         task_id='prepare',
3         bash_command='matlab -noFigureWindows -sd {{
          ↪  var.json.ap_cfg.project_dir }} -batch \
4         addpath(genpath(pwd)); \
5         srcs = base64_to_mat(\'{{ dag_run.conf.map.source_file }}\');
  ↪  \
6         src = srcs({{ dag_run.conf.model.source }}, :); \
7         rcvrs = base64_to_mat(\'{{ dag_run.conf.map.receiver_file
  ↪  }}\'); \
8         rcvr = rcvrs({{ dag_run.conf.model.receiver }}, :); \
9         prepare_ram({{ dag_run.conf.model.model_choice.RAM.freq }},
  ↪  src(3), rcvr(3), \'ram.in\');"',
10        dag=ArcticOceanDag
11    )
```

Listing 5.1: The Bash operator code for preparing the RAM input files.

Listing 5.2 contains an example of a Bash operator used to run the RAM model. The Bash task first navigates to the RAM acoustic model folder and then starts the script `launch_ram.sh`, which builds the RAM image if none exist and then runs the RAM model within a Docker container.

```
1     run_ram = BashOperator(
2         task_id='run',
3         bash_command='cd {{ var.json.ap_cfg.project_dir }}/{{
          ↪  var.json.ap_cfg.models_dir }}/RAM; bash launch_ram.sh ',
4         dag=ArcticOceanDag
5     )
```

Listing 5.2: The Bash operator code for running the RAM model. The launch_ram script is shown in Listing 4.4. Each model has its own launch script available in [74].

5.2.2 BranchPythonOperator

The `BranchPythonOperator` selects branches based on the return value of the `python_callable` parameter. As can be seen in code Listing 5.3, the `BranchPythonOperator` sets the Python callable argument to be the function `select_model` and the workflow context is passed to the function. The function then finds `model_choice` from the context in the JSON input, and checks which models are chosen. Those model choices gets added to the return array. For Bellhop the model type is also checked, as each model type is a separate branch in the workflow.

```
1    def select_model(**context):
2        model_selection =
         ↪  context['dag_run'].conf['model']['model_choice']
3        branches = []
4        if model_selection['run_ram']:
5            branches.append('ram.prepare')
6        if  model_selection['run_mpiram']:
7            branches.append('mpiram.prepare')
8        if model_selection['run_bellhop']:
9           bellhop_type = model_selection['Bellhop']['simtype']
10          for simtype in bellhop_type:
11               branches.append(f'bellhop_{simtype}.prepare')
12       if  model_selection['run_eigenray']:
13           branches.append('eigenray.prepare')
14       return branches
15
16   branching = BranchPythonOperator(
17       task_id='select_models',
18       python_callable=select_model,
19       dag=ArcticOceanDag
20   )
```

Listing 5.3: The Airflow implementation of the branching, where the `BranchPythonOperator` selects which model branches to run.

5.2.3 Task Groups

The main purpose of task groups is to group related tasks together and reduce visual clutter. In the Airflow webserver, the task groups are shown as one node, but can be expanded to show the tasks within the task groups. This is shown for the RAM model in Figure 5.2.

For the Bellhop model, five almost identical branches are created and executed based on which model options were selected by the user. The first attempt at handling the five different model options, was to have only one branch that could run multiple times and have the model type as an input argument. The reason this

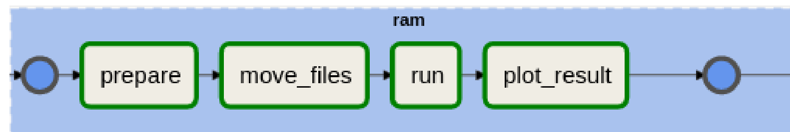


Figure 5.2: The expanded view of the RAM task group from Figure 5.1

is not possible, is that Airflow generates a Directed **Acyclic** Graph (DAG), which means that jumping from the last Bellhop node and back to the first, is not possible as it creates a cycle. An option could be to generate tasks dynamically based on the user input, but it is simpler and clearer to generate one branch for each of the types and skip the branches that are not selected by the user. To generate the five bellhop branches, a for-loop was used to avoid copy-pasting the task group for each of the model types. Listing 5.4 shows how one branch for each model type was created: Eigenray (E), Ray (R), Coherent transmission loss (C), Semi-coherent transmission loss (S), and Incoherent transmission loss (I).

```
1     for simtype in ["E", "R", "C", "S", "I"]:
2         with TaskGroup(f'bellhop_{simtype}') as bellhop_model:
3             prepare_bellhop = BashOperator(
4               # Operator code
5             )
6 
7             run_bellhop = BashOperator(
8                 # Operator code
9             )
10
11            plot_bellhop_result = BashOperator(
12                # Operator code
13            )
```

Listing 5.4: The code to create a branch for each model type in the Bellhop acoustic model.

5.3 REST API

Section 2.3.2 shortly introduced the REST API included with Airflow. The API supports Create, Read, Update, and Delete (CRUD) operations on the Airflow metadata objects, such as DAGs, DAG runs, and variables. The API is ordered by the resources, and the endpoint naming is plural and camel cased (CamelCase). Field names are written in snake case (snake_case).

The Arctic Package Airflow workflow, sends a request to the endpoint at `api/v1/dags/{dag_id}/dagRuns` to start a workflow run. The request is a POST request where the payload is the user input from the GUI. An example payload is shown in Listing 5.5. It is also possible to specify the identifier for the DAG run,

```
{
    "conf": {
        "map": {
            "minlon": -180,
            "maxlon": 180,
            (...)
            "data_selection": {
                "color_db": "IBCAO 2min"
                (...)
            }
        },
        "model": {
            (...)
            "model_choice":{
                "run_mpiram": true,
                "run_ram": false,
                "run_bellhop": false,
                "run_eigenray": false,
                "MPIRAM": {
                    "freq": 50,
                    "q_value": 5,
                    "time_window": 10
                }
            }
        }
    }
}
```

Listing 5.5: An example of a request body to trigger a DAG. The `conf` object is generated by RSJF, and the Airflow workflow requires that the input is named and structured as in this example, otherwise the Airflow workflow will not recognize the variables. The (...) section indicate that a part of the object was left out for brevity.

but if it is not specified it is automatically generated based on the execution date. If the user tries to trigger a DAG run with an existing ID, it fails with a `409 Already exists` error.

5.4 MATLAB Adaptations for the Airflow workflow

Section 4.3 described how the Arctic Package MATLAB code was adapted to the KNIME workflow. The project was modularized by dividing it into smaller parts. The new structure worked well with the KNIME workflow, but is also suitable for the Airflow workflow. The modular and smaller functions could easily be divided into Airflow tasks and task groups. However, some further adjustments to the Bellhop

model were needed to achieve the same functionality as in the KNIME workflow.

The main change was to the Bellhop model. It consisted of adding a new parameter for specifying the filenames of the input and output files to the `prepare_bellhop_input()` function and the `plot_bellhop()` function. This was because the Bellhop model can run multiple different model types in one workflow run, e.g. rays and eigenrays. However, if multiple Bellhop models are selected in the same Airflow workflow run, there is no specified order for when the tasks within the model type branches are run relative to each other.

For example, the Bellhop model task group has three subtasks: `prepare_bellhop`, `run_bellhop`, and `plot_bellhop_result`. The Arctic Package prepares an input file with the name `belltemp.env`, regardless of which Bellhop model type is run. The models also produce the same output files to the same location, which would result in the files being overwritten. The result is that all model types produce the same result, unless the tasks executes sequentially and one task group completes before the other starts. When each model type has its own specific name for the environment file e.g. `belltemp_R` for ray and `belltemp_E` for Eigenray, the files do not overwrite each other and the plotting function knows which result files to use.

The Bellhop change required a change in the Dockerfile for building and running the Fortran Bellhop model and in the script that launches the Docker container. The launch script changed the command to run a Docker container to be the environment name inputted to the launch script. Listing 5.6 show the original command used to run the Docker container in line 1 and the new command in line 3. The change to the Dockerfile is similar, and replaces every occurrence of `belltemp`, which was the original environment name, with `$SIMNAME` that is inputted to the launch script.

```
docker run --name bellhop_container --mount type=bind,
    source=/$(pwd)/data,target=/model/data bellhop

docker run --name bellhop_container -e SIMNAME=$1 --mount
    type=bind,source=/$(pwd)/data,target=/model/data bellhop
```

Listing 5.6: Docker commands to start the Docker container that runs the Bellhop model.

After implementing the changes to adapt the Arctic Package to the Airflow model, the nodes in KNIME were updated to specify the environment name `belltemp` in the nodes that previously did not. The same components were still used in KNIME and only required the change of adding the environment name to the function calls for preparing the data for modeling and reading the results.

In addition to the Bellhop change, a MATLAB function to convert Base64 to a MATLAB matrix format was added. This was necessary as the RJSF GUI encodes the source and receiver files selected by the user with Base64 before passing them

on as input to the Airflow workflow. The Base64 encoded files has to be converted back to the matrix format for MATLAB to be able to read the contents of the files. The function is called `base64_to_mat()` and was added to a separate folder called `airflow`.

Chapter 6

DevOps Pipeline

An important aspect of improving maintainability and portability in the Arctic Package from a software development perspective, is to establish a DevOps pipeline for the package. This chapter describes the existing pipeline for the Arctic Package in Section 6.1 and Section 6.2 describes the updated pipeline for the KNIME workflow and the Airflow workflow and how to set up and update the workflow versions of the Arctic Package.

6.1 Existing Arctic Package Pipeline

The original version of the Arctic Package did not have an established DevOps pipeline. The source code was distributed from user to user without any structured version control. In addition, the library `m_map` and the acoustic model implementations were not kept up to date, as they included code in zipped files that were copied into the project. The data the models use were also being passed from user to user as text files together with the zipped source code. As a consequence, multiple different versions of the package is in use. This shortcoming can be addressed by creating a DevOps pipeline for the project. The source code should be under source/version control, which can be achieved by uploading the code on GitHub, or another source control platform along with all further updates to the code.

Another challenge with the pipeline, is that the acoustic models implemented in Fortran must be compiled and built and an environment for this has to be established. It can be useful to containerize the models, as the Fortran source code for the models require different steps and software for compiling and running the models on different operating systems. If the models are compiled and run in a container, for example a Docker container [51], the models are run in the same environment every time and can be deployed on all operating systems. This makes the behaviour predictable and the models easier to run.

6.2 Updated Pipeline

The workflow version of the Arctic Package includes an updated pipeline. Instead of having all components in a zipped file and sending it to users that need it, the

source code is split up and kept in four GitHub repositories:

- A MATLAB source code repository, containing source code for processing of data and plotting of model results [75].
- An acoustic model repository [74], containing the source code for the Acoustic models. The repository also contains a Dockerfile for each model and a Bash script to build and run the model in a Docker container. Section 6.2.1 describes Docker and the Dockerfiles that was developed in more detail.
- Workflow repository, containing the KNIME workflow model [76].
- Workflow repository, containing the Airflow workflow model [77].

The GitHub repositories containing the Airflow workflow and the KNIME workflow are public, as they are created by the author of this thesis. However, the repositories for the acoustic models and the MATLAB source code are both kept private in order ensure that copyright is not infringed. Access can be granted on request.

When setting up the Arctic Package, it is only necessary to use either the Airflow workflow or the KNIME workflow, depending on which of the workflows that is preferred. In addition to the GitHub repositories, there are datafiles containing sound speed, bathymetry, salinity, source information, and receiver information. These files are not at this time kept in a shared repository or database, but that is encouraged for the future. The files were too large to keep in a free GitHub repository, and creating a shared database for the files would require investigating different options and setting up connections to the database from the workflows. This was not part of the requirements, and time spent on this would mean that less of the requirements would be finished.

The steps from code change to a new version of the software, depends on which part of the package is changed. The two steps that are always included is to commit and push the code changes to the GitHub repository. Additionally, it is useful to pull the latest updates from the repository before making changes.

The new pipeline described in this section simplifies distribution the Arctic Package and make sure that all users have access to the most recent version. The pipeline does not handle how to update the `m_map` library and the acoustic models when they are updated by the authors, as they are available at the authors website and must be downloaded. Instead, the version used by the Arctic Package is available in a repository to make sure that all users have the same version of the software. The workflow versions of the Arctic Package also makes it easy to compile and run the Fortran code, as it is run within a Docker container.

The rest of this chapter describes the pipeline for the workflow versions of the Arctic Package and how to set it up on a local computer. Section 6.2.1 describes the process of updating the acoustic models, Section 6.2.2 describes the MATLAB source code pipeline, Section 6.2.3 describes the pipeline, setup, and technology stack for the KNIME workflow, Section 6.2.4 describes the pipeline, setup, and technology stack for the Airflow workflow. Some of the software in the technology stack is common

for both workflows: MATLAB, Docker, and Python. The pipeline does not at this point include deployment to a server, but that can be added later if required.

6.2.1 Acoustic Models Pipeline

The acoustic models are compiled and run in a Docker container. The containers ensures that the application is compiled and run in the same environment every time, no matter which host is used.

Docker

Docker is a platform to develop, ship, and run applications in an isolated environment called a container. The containers do not rely on the software installed on the host, but instead has their own environment. Docker containers are created from Docker images. A Docker image is a read-only template that can be used to create containers from. Docker containers are runnable instances of an image. The containers can be created, started, stopped, moved, or deleted.

The Docker images are built automatically by reading the instructions in the Dockerfile. The Dockerfile for the RAM acoustic model is shown in Listing 6.1. Line 1 shows that the RAM Docker image is created from the latest version of the image `centos`. Line 5 installs the gfortran compiler and the make utility. Line 7 copies the source files for the RAM acoustic models, and line 8 sets the working directory for the container. Line 9 runs the make command, that compiles the Fortran source code and creates the executable `ram.exe`. Line 11 shows the command that is run in the container when the command `docker run` is executed.

```
1   FROM centos:latest
2   RUN yum update -y
3
4   # Add fortran
5   RUN yum install -y gcc-gfortran make
6
7   COPY makefile ram.f /ram/
8   WORKDIR /ram/
9   RUN make ram.exe
10
11  CMD ["sh", "-c", "cp /ram/data/ram.in /ram/ram.in &&
     ↪  ./ram.exe && mv /ram/{tl.grid,tl.line} /ram/data/"]
```

Listing 6.1: The contents of the Dockerfile for the RAM acoustic model.

Updating the Acoustic Models

To change the acoustic model code, the developer needs to follow four steps:

1. If a new file is added, add it to the Makefile, so it will be compiled. If an existing file is changed, skip this step.

2. Update the Docker container the acoustic model is run in:

   - Delete the existing Docker container for the acoustic model using the command `docker rm <container_name>`, as the container does not contain the new changes. Existing containers can be shown with the command `docker ps -a`.
   - Check the Dockerfile to make sure the new file is included to the Docker image when it is built (line 7 of the Dockerfile shown in Listing 6.1).
   - Rebuild the Docker image of the acoustic model. The command `docker build -t <image> .` is used to rebuild a Docker image. The `-t` option makes it possible to specify the image tag of the image to rebuild. The `.` specifies the path where the Dockerfile is, in this case it is located in the current directory.

3. If the change included changes to the input or output parameters, change the MATLAB data processing code to make sure the input files have the correct format, and that the output files can be read by the MATLAB functions that plot the simulation results.

4. Push all changes to the repository.

6.2.2 MATLAB Source Code Pipeline

To change the MATLAB source code, the developer only needs to push the code to the repository and make sure the MATLAB code is still compatible with the workflow and acoustic models. For example, if the input or output parameters of a function was changed, the workflow's call to the function must be updated to include the new parameter.

An example is if the parameter `depth` was added to the `plot_ram` function. Listing 6.2 shows how the configuration view of the node calling the function would be implemented. Listing 4.6 from Section 4.2, shows the version without the change. The change is in line 9, where `plot_ram(res.freq, res.zs, res.zr, res.r, res.z, res.ttRAM, res.rdRAM, res.rrRAM)` has changed to add `res.depth` to the end function call.

6.2.3 KNIME Workflow Pipeline

To set up the KNIME workflow, the following software is needed in addition to the repositories listed in the start of Section 6.2:

- Python, tested with Python 3.7 [78]
- The KNIME Analytics Platform, tested with version 4.3 [49]
- Docker [79]
- MATLAB, tested with MATLAB R2020b [80]

```
1     matlablib = strcat(kIn.knimeworkspace, kIn.matlablib);
2     addpath(strcat(matlablib, '/global'));
3     addpath(strcat(matlablib, '/models/RAM'));
4
5     % Load RAM results
6     load(kIn.save_filename, 'res');
7
8     % Plot RAM results
9     plot_ram(res.freq, res.zs, res.zr, res.r, res.z, res.ttRAM,
       ↪  res.rdRAM, res.rrRAM, res.depth);
```

Listing 6.2: The configuration code for the `Plot RAM` node in Figure 4.6.

To set up the KNIME workflow, install the software technology stack and import the KNIME archive to the KNIME workspace. When the workflow is opened, a prompt asking to install the missing extensions appears. Press ”yes”, as the extensions are needed in the workflow. These should be the community extensions Vernalis KNIME Nodes [81] and KNIME Matlab Scripting extension [52], and the KNIME extension KNIME Python Integration.

When the workflow is imported, the GitHub repositories and the datafiles must be set up. In order for KNIME to locate the files, create a folder in the imported workflow called ArcticOceanFiles. This is the folder where KNIME will look for the files that the workflow needs to run. If the files are located at another location, the `Configure workflow folder paths` node in the workflow must be reconfigured to the correct path. Clone the Arctic.Package-v2 and AcousticModels repositories into the specified folder. Create the folders `dataBase`, `rigFiles`, and `savedModels` and place the datafiles into the `dataBase` folder and the source and receiver files in the `rigFiles` folder. If any of the above folders are placed at another location, reconfigure the `Configure workflow folder paths` node to the correct path. KNIME should look something like in Figure 6.1 after completing the setup. The top left box in the figure shows the suggested file structure.

For the future, the setup could be automated by creating a setup script for the workflow. The script could clone the required repositories and set up the default file structure. This can reduce the number of possible errors to make by a user or developer when setting up the Arctic Package.

To run the workflow, set the input parameters in the first node, `Take input parameters` in Figure 6.1, and for the acoustic models. To give input to the workflow click on the `Take input parameters` node and press `shift + F10`. A GUI that asks for user input should appear. To give input to the acoustic models, open the component node for the model to reveal the input node, click on it and press `shift + F10`. A view to set the input parameters should appear. For the MPIRAM model, the view is shown in Figure 6.2. Note that for this approach, the dependencies for the model component needs to be executed. Another approach is to press `shift`

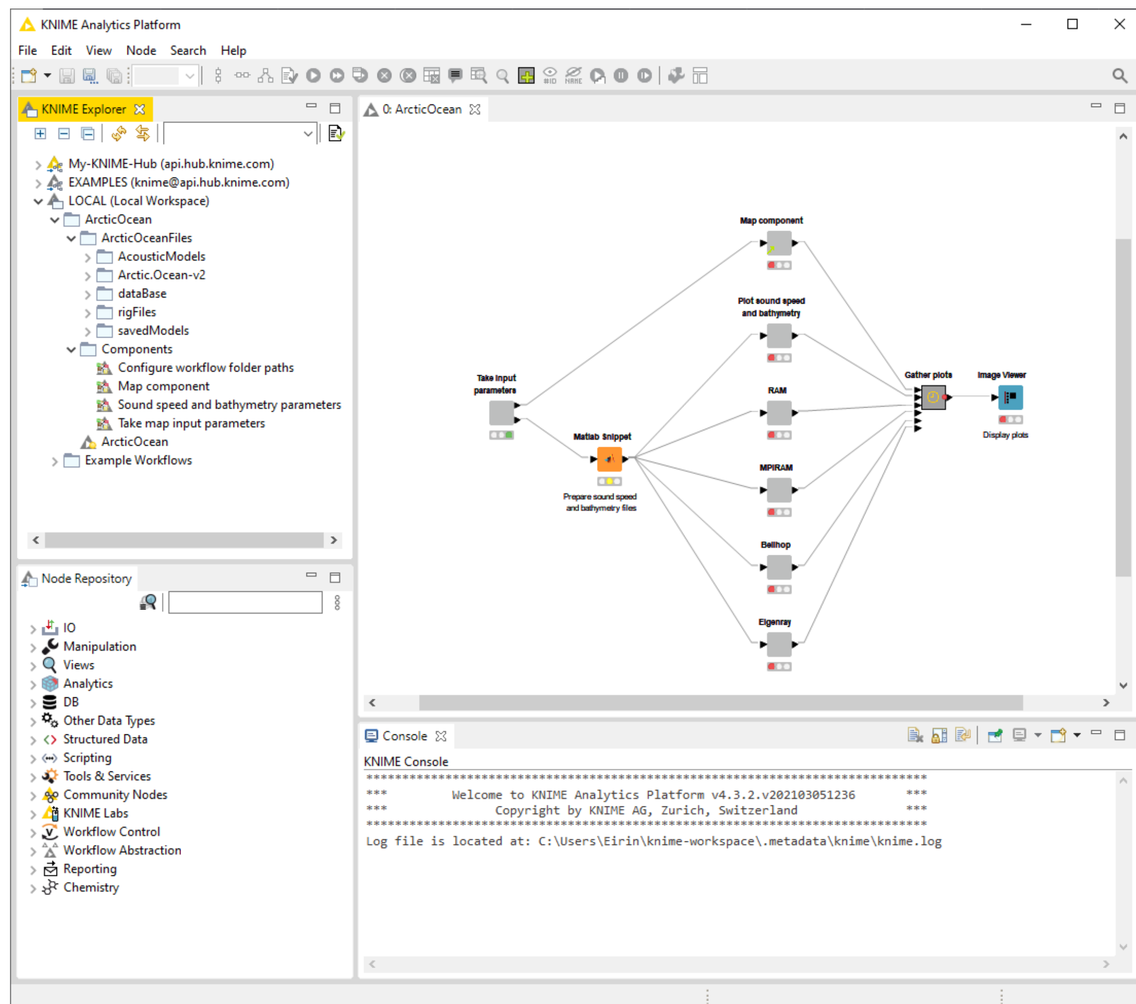


Figure 6.1: The KNIME tool view after successfully setting up the KNIME workflow. The file structure is shown in the left box named `KNIME Explorer`, the KNIME node options are shown in the `Node Repository` box, and the workflow is shown in the open tab: `ArcticOcean`.

+ F10 for the model components at the top level of the workflow, but that will run the models with the default values first and then open the input view, which results in running the workflow twice to get the results. An alternative way to open the input view is to right click on a node and click `Interactive View: <node name>`, as shown for the `Take MPIRAM input` node in Figure 6.3. After setting the input, all nodes can be run by pressing `shift + F7`. A specific node can be run by clicking on it and pressing `F7`. All dependencies of that node runs first and then the node is executed. Section 7.2 contains more details about the KNIME Workflow GUI.

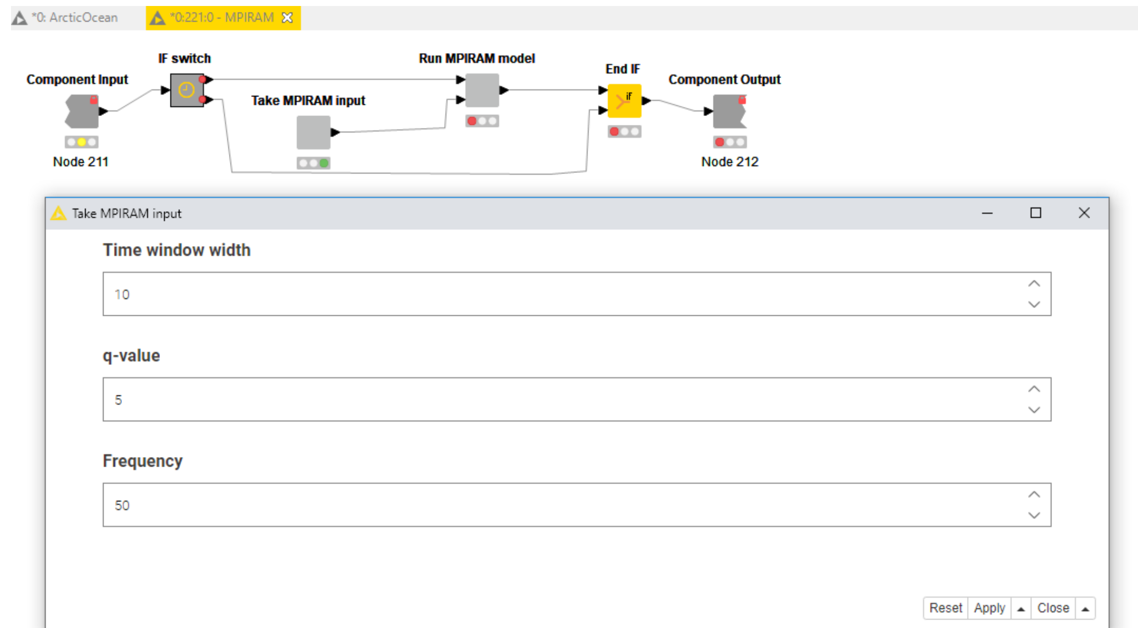


Figure 6.2: The expanded view of the MPIRAM node in Figure 6.1 with the generated input view. The `Take MPIRAM input` node has been run, indicated by the green circle below the node, and is ready to take input.

When the KNIME workflow is changed, make sure that if a shared node was updated, the update is fetched by the workflow node by right clicking on the node linked to a shared component and select `update link`. When all changes are made, export the workflow and push it to the repository.

6.2.4 Airflow Pipeline

To set up the Airflow workflow, the following software is needed in addition to the repositories listed in the start of Section 6.2:

1. Python, tested with Python 3.6.9 [78]
2. MATLAB, tested with MATLAB R2020 [80]
3. Docker [79]
4. Airflow $\geq$ 2.0 [44]

For the GUI (to be discussed in Chapter 7):

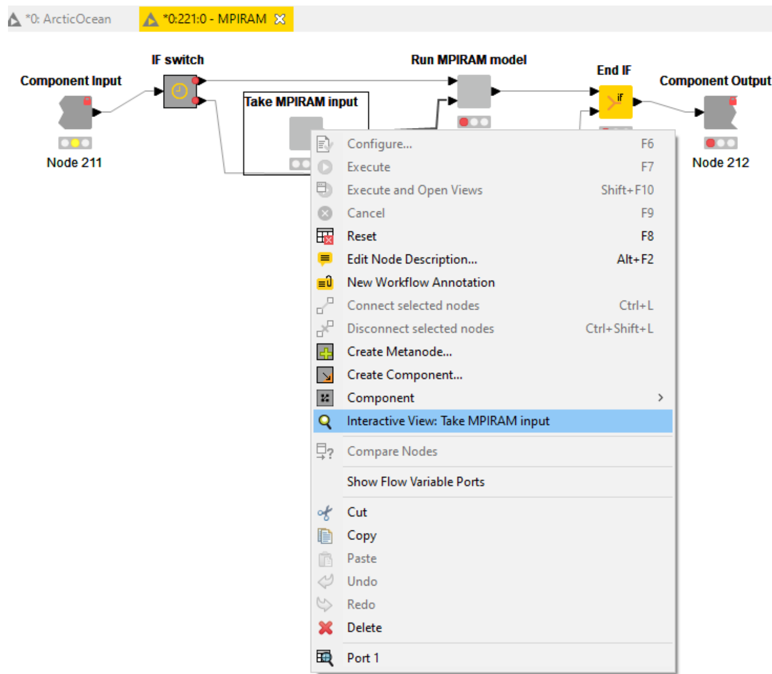


Figure 6.3: An alternative way to open the input view in Figure 6.2.

1. Node ≥ 10.16 and npm ≥ 5.6 [82]

To set up Airflow, first install the software technology stack and then clone the repositories and acquire the datafiles. Put the datafiles and repositories, except for the Airflow workflow repository, into a folder named `~/ArcticOcean`. The Airflow workflow searches for the Arctic Package files in that folder. If the folder is named something else, the `settings.json` file in the Airflow-workflow repository needs to be changed accordingly. An example of a `setting.json` file is shown in Listing 6.3. The suggested file structure looks like this:

```
user
├── Airflow-workflow
│   ├── arctic-ocean-gui
│   └── dags
│       ├── settings.json
│       └── arctic_ocean.py
└── ArcticOcean
    ├── Arctic.Ocean-v2
    ├── AcousticModels
    ├── dataBase
    ├── savedResults
    └── rigFiles
```

Check that the file names and locations for the Arctic Package workflow matches the folder names and locations in your `settings.json` file. If the locations or naming

```
{
    "ap_cfg":{
        "project_dir": "~/ArcticOcean",
        "matlab_lib":"Arctic.Ocean-v2",
        "models_dir":"AcousticModels",
        "database_dir":"dataBase",
        "save_dir": "savedResults"
    }
}
```

Listing 6.3: The settings file containing path information about the Arctic Package folders.

in the file system is different from the file, update the settings file.

To install Airflow, follow the installation instructions at the Airflow documentation site [44]. Before installation, set the environment variable `AIRFLOW_HOME` to be the path to where the Airflow-repository was cloned to, for example `export AIRFLOW_HOME=~/Airflow-workflow`. The default value is `~/airflow`. When the installation is complete, initialize the database with the command `airflow db init` and create a user with the command shown in Listing 6.4. The different roles available are:

- Admin, that has all possible permissions.
- Public, that are anonymous users that do not have any permissions.
- Viewers, that have a limited set of permissions. The permissions are related to viewing information about the DAGs.
- User, which have viewer permission and permission to manage the DAGs.
- Op, which have User permissions and permission to import variables and configure workflows.

```
airflow users create \
    --username <username> \
    --firstname <your firstname> \
    --lastname <your lastname> \
    --role <Admin, Public, Viewer, User, or Op> \
    --email <your-email>
```

Listing 6.4: Command for creating an Airflow user. Replace the angle brackets with the information of the user to be created.

Airflow creates a configuration file `airflow.cfg` the first time Airflow is run. The file is located in your `$AIRFLOW_HOME` directory. A reference configuration file used

for this project is available at the workflow repository [77]. The configuration file can be edited to change Airflow's default settings. The default setting for requests to the REST-API is to deny all requests. For the GUI to be able to send requests to the REST-API, the setting must be changed to the preferred authentication method. For local use, it is possible to set it to `default` which disables authentication completely. However, this should not be done for production use where others have access to the API. To change the setting, locate the `airflow.cfg` file and change `auth_backend = airflow.api.auth.backend.deny_all` to `auth_backend = airflow.api.auth.backend.default`.

Start the webserver with the command `airflow webserver --port 8080`. It is important that it is port 8080 as the GUI searches for the Airflow REST API at port 8080. Start the scheduler with the command `airflow scheduler`. Now the webserver should run at localhost:8080.

Import the `settings.json` file as a variable to Airflow through the CLI: `airflow variables import settings.json` or import the variables through `Admin` $->$ `Variables` in the Airflow webserver. The variables are used by the workflow to locate the source code in the Arctic Package.

To start the GUI, first install the dependencies with `npm install` then run `npm start`, which launches a development server on localhost:3000. `npm run build` can be used to create a build for production to the `build` folder.

To run the workflow, go to localhost:3000, fill out the parameters with the desired values and check at localhost:8080 to see the progress of the workflow run. Do not start more than one DAG-run at the same time, as the processes would interfere with each other, because there are files created with each DAG run, and the names of those to not change.

To change the Airflow workflow, change the `arctic_ocean.py` workflow file in the Airflow Python project. Check the workflow file for errors by running `python <workflow_file>`. The tasks can also be tested by running `airflow tasks test <dag> <task> <date>`, but the command does not take user input, which means that the command will not work for tasks using input from the GUI.

If new input parameters are added or a parameter needs to change due to workflow changes, the GUI for Airflow must be changed accordingly to ensure that the GUI takes the same input parameters as used in the workflow. The GUI is a React application, which uses the library React-JSONSchema-form [83] to generate a GUI from a JSON schema that defines the input parameters. The GUI is described in further detail in Section 7.3.

The JSON schema is set up as in Listing 6.5, where each parameter has a title, a type, and a default value. There are other options to be set in the schema as well, but the ones shown in the example are the most used. The option `type` is the only option that is required. To add a parameter to the schema, create a new parameter with a title, type, and a default value, and add it to the list of parameters in the

```
1        "radius": {
2            "title": "Map radius",
3            "type": "integer",
4            "default": 15,
5        },
6        "shape": {
7            "title": "Map shape",
8            "type": "string",
9            "default": "rectangular",
10           "enum": [
11               "circular",
12               "rectangular"
13           ],
14           "enumNames": [
15               "Circular",
16               "Rectangular",
17           ]
18       },
19       "source_file": {
20           "title": "Select file to load sources from",
21           "description": "Select file to load sources from",
22           "type": "string",
23           "format": "data-url",
24       }
```

Listing 6.5: An example from the JSON schema used to generate the GUI for Airflow. The `radius` parameter is an integer parameter with a default value of 15. The `shape` parameter is a string parameter with two choices, `circular` and `rectangular`. The `enumNames` setting sets the display names for the enum values, while the enum values are the values sent to the workflow. The `source_file` parameter, is a string parameter that takes a file as input and encodes it as base64 before sending it to the workflow.

JSON schema. To see the changes in the GUI, start the GUI on a local development server with `npm start`.

When all changes to the Airflow workflow are done, push the changes to the Airflow workflow repository.

Chapter 7

Generated User Interfaces

The original MATLAB GUI for the Arctic Package is not used with the SWFMSs. Integration with a SWFMS makes it possible to decrease the amount of MATLAB code, as a portion of the code is centered around the GUI and GUI functionality. Our goal is to automatically generate a GUI based on a set of model parameters instead. When automatically generating a GUI, it is easy to add new parameters to the models. The only change that must be made is to add the parameter definition and the GUI will be automatically updated. This can improve the maintainability, as no new code has to be written when including new input parameters. However, the input parameters for the GUI still needs to be specified.

The original GUI was created in MATLAB and is described in Section 7.1. KNIME has a built-in way to handle GUIs based on a type of nodes called widget nodes. The KNIME GUI generation is described in Section 7.2. Airflow does not have a built in way to generate a user interface, but instead a React project using the React-JSONSchema-Form (RJSF) library [83] was created. RJSF automatically generates a GUI from a JSON Schema specification of the input parameters. The Airflow GUI is described in Section 7.3.

7.1 Original GUI

The original GUI was implemented in MATLAB, where each of the parameters and buttons were added to the GUI through MATLAB scripts. In total, there were 28 files that provided functionality to draw or change the GUI. That amounts to 1518 Lines of Code (LoC) counted by CLOC [68], not including comments and blank lines. This count also excludes all the lines that retrieves the input values from the GUI. Due to the strong coupling between the GUI and the data processing, those lines are hard to count, as they would have to be manually located and counted throughout the package.

The original GUI of the Arctic Package includes an interactive map that the user uses to select sources and receivers for modeling. The GUI is shown in Figure 7.1 and the interactive map in Figure 7.2. There are new GUI windows that appear when the GUI is interacted with. For example, selecting a model under the `Choose Model` tab and clicking `GET PREDICT`, creates a separate window with the model

configuration parameters. Figure 7.3 shows the model configuration window for the Eigenray model. The button `Advanced Src&Rcvr Edit` makes the advanced source and receiver edit GUI in Figure 7.4 pop up in a separate figure from the map and the main GUI. The button `Visualizations of saved results` makes the result plotter in Figure 7.5 pop up in a separate window.

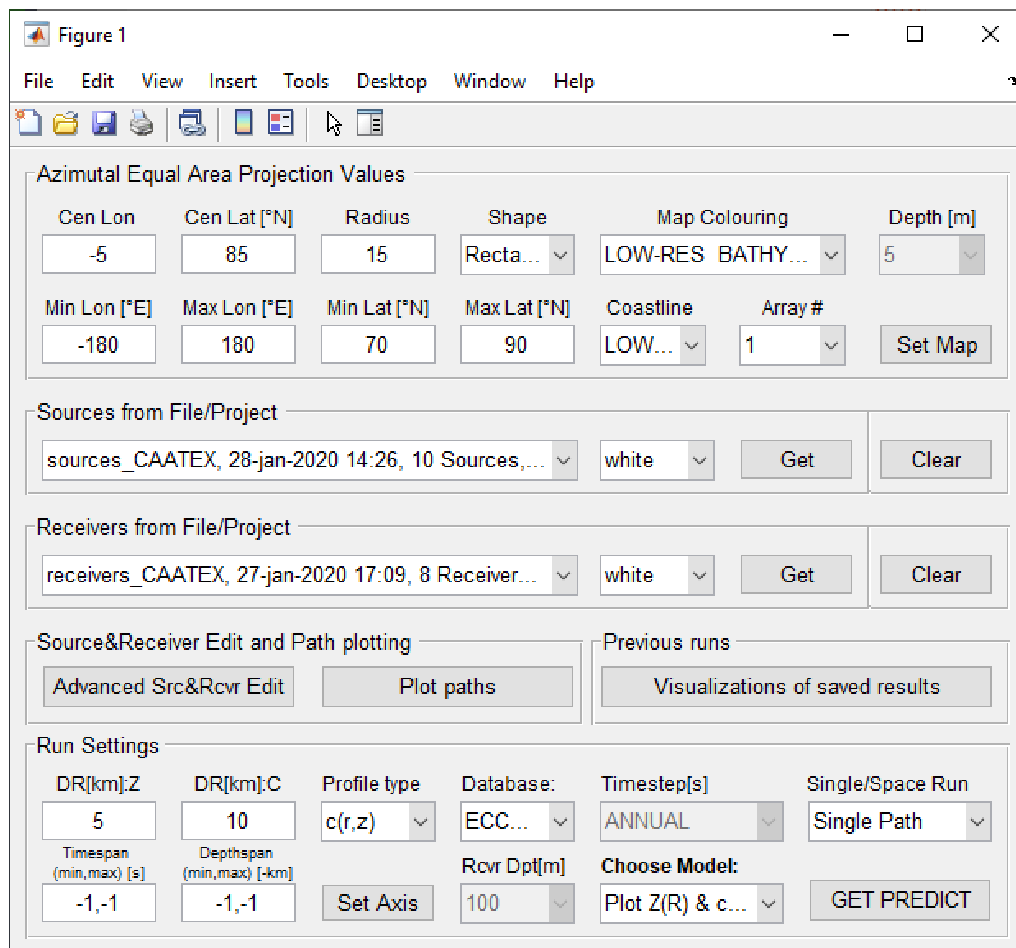


Figure 7.1: The original GUI in the Arctic Package. It takes user input for creating the map (Figure 7.2) and for acoustic modeling.

Another aspect of the original GUI, is that the user sometimes has to interact with the MATLAB console instead of the GUI or the map. For example, when the `GET PREDICT` button in Figure 7.1 is clicked, a message `Click near the Desired Source` appears in the console. There are no instructions in the GUI that indicate that clicking on a source in the map is the next step, and it appears that nothing happens when the button is clicked. The same issue occurs when the `Clear` button is clicked. A message `Removing all sources; hit return to continue or ^C to abort` appears in the console, and the only way to confirm or abort is to click the MATLAB console and either press return or ˆC. All error messages are printed to the MATLAB console, and there are no checks for missing values. If the user forgets to load the sources to the map, the error message in the MATLAB console is: `Unrecognized function or variable 'srcs'.`, but no errors are shown in the GUI.

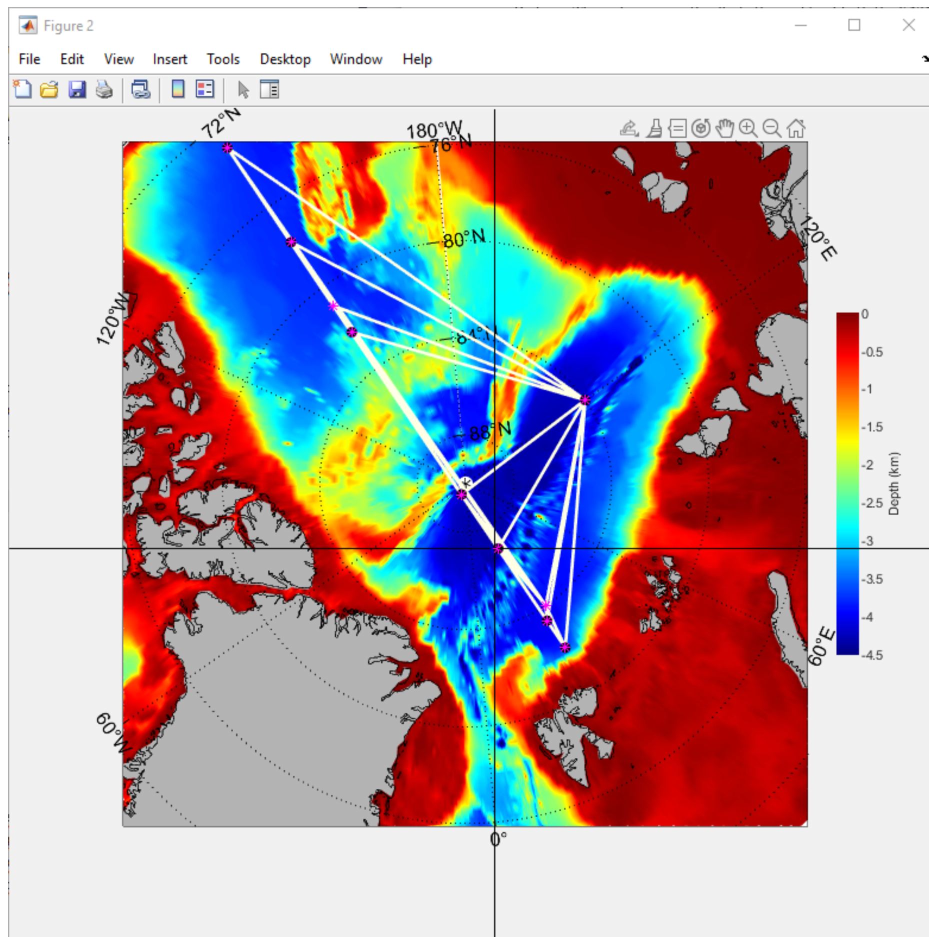


Figure 7.2: The interactive map in the Arctic package. It is used to select sources and receivers and to view the coverage of different databases over a specified area. The intersection between the black lines, indicates the cursor placement. This is the same map as in Figure 3.3.

Eigenray Parameters

Eigenray Parameters

Eigenrays | Save Paths

No. of Rays 2500 | Use bottom

Epsilon 1e-5 | Angle Range -15, 15

Max # Bot Refl 3

GET PREDICT

Figure 7.3: The GUI window for configuring the Eigenray parameters. When `GET PREDICT` is clicked, the user has to select a source and receiver from the map.

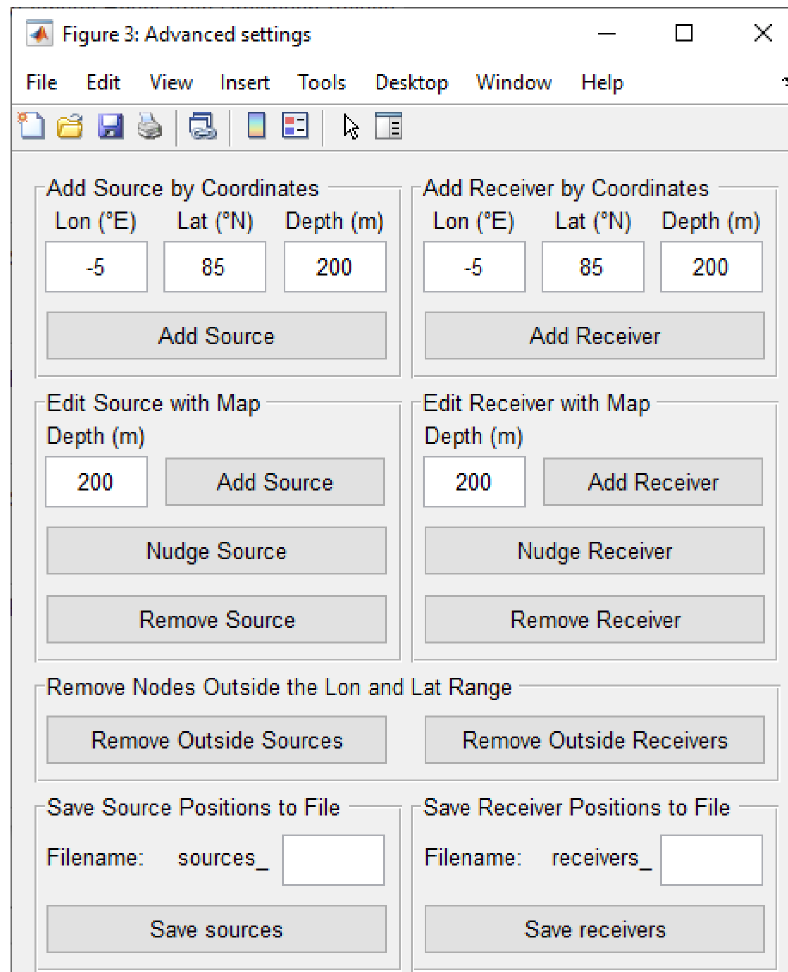


Figure 7.4: The advanced source and receiver GUI window. The user can manually add sources and receivers, nudge, remove, or save them to file. When a source or receiver is nudged, the user can move it from one placement on the map to another using the cursor.

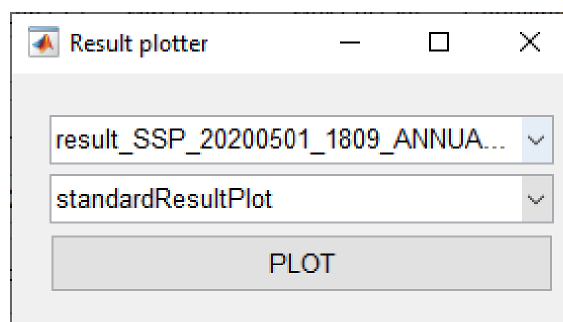


Figure 7.5: The result plotter that allows the user to plot previously saved results to a figure. Some of the saved plots contain multiple different options for plotting that can be selected in the bottom drop-down.

7.2 KNIME Built-in GUI

KNIME has a built-in feature to automatically generate a GUI. The generated GUI passes the user input to the workflow as workflow variables. Widget nodes are a special type of nodes in KNIME that can be used to generate a GUI. The different types of widget nodes used in the KNIME workflow are:

- Integer widgets, which takes a single integer. Minimum and maximum values can be set.
- String widgets, which can take single- or multi-line input. The input can be validated against a regular expression.
- Boolean widgets, shown as a checkbox and outputs true if the box is checked and false otherwise.
- Single selection widget, where the user selects a value from a set of specified values. The values can be shown as a drop-down menu, a list, or radio buttons. A list provides the values under each other in a box in the GUI, and the user marks the selected element.
- Multiple selection widget, where the user can select any number of elements from a set of values. The values can be shown as checkboxes, a list, or a twinlist. A twinlist shows two boxes of elements, one of values to be included (selected) and one with the elements to be excluded (not selected).

Figure 7.6 shows a user interface generated by KNIME, and Figure 7.7 shows the widget nodes that are used to generate the GUI. All of the widgets have default values set for the parameters, which makes it possible to avoid using the KNIME generated GUI altogether, by just setting the default values to the desired values in the widgets.

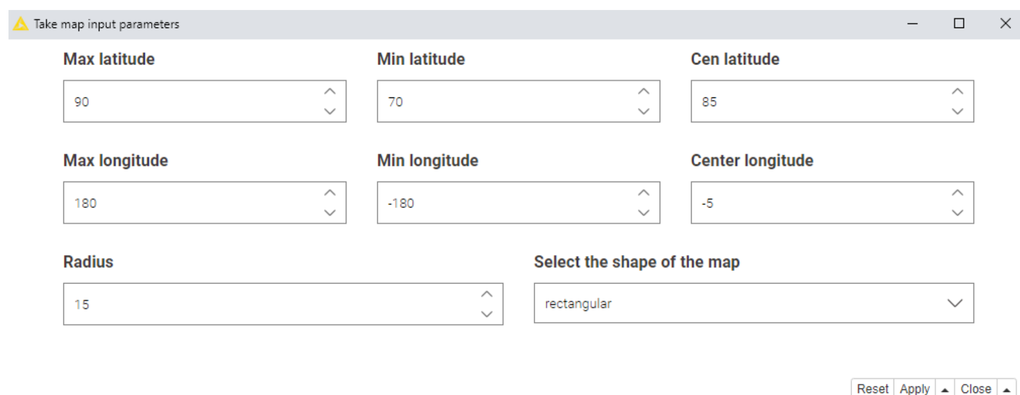


Figure 7.6: The GUI generated by KNIME from widget nodes in Figure 7.7 specifying the map parameters.

To generate the GUI for the map input parameters, integer nodes were used for all parameters, except for the shape parameter, which uses a selection widget. The selection widget is configured to be shown as a drop-down menu. When the widget

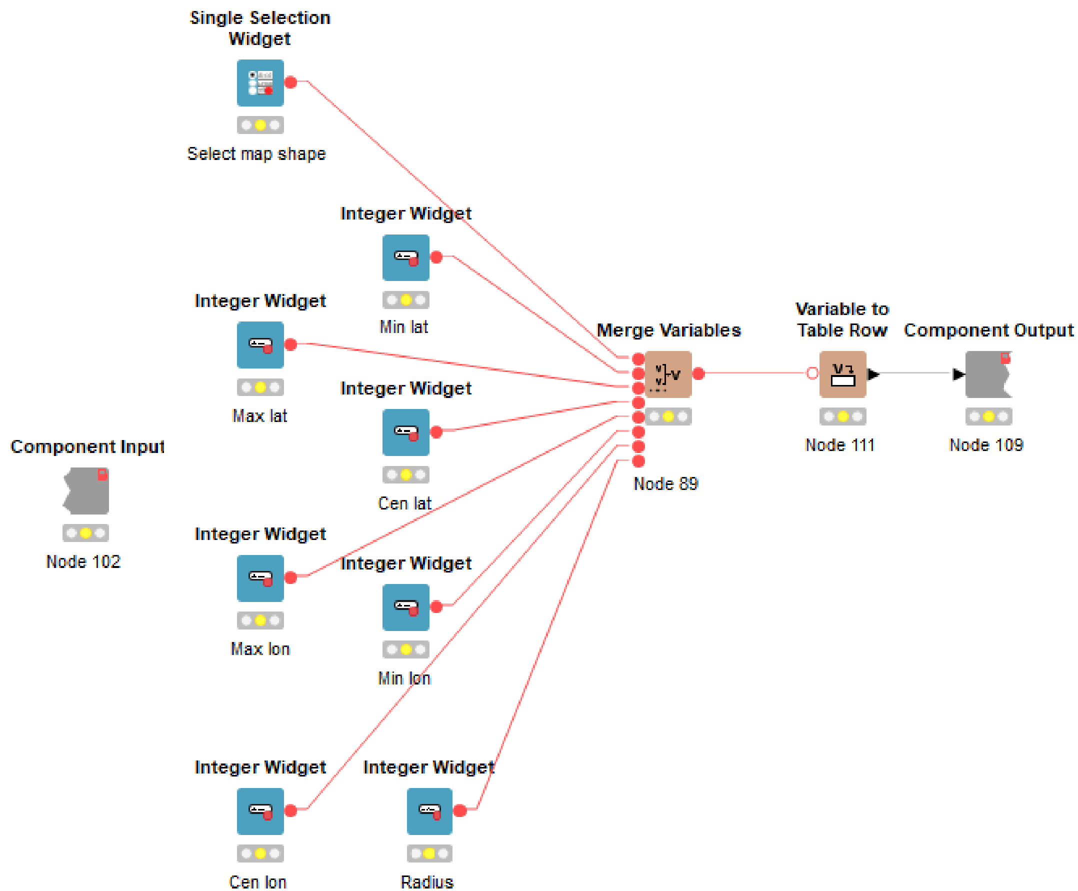


Figure 7.7: The `Take map input parameters` node in the KNIME workflow. The node consists of widgets to take user input. The integer widget takes integer input and single selection widget creates a drop-down for the user. This figure is a copy of Figure 4.3.

nodes are grouped together in a component, the GUI can be organized by dragging and dropping the parameters to a desired position in the GUI, as shown by Figure 7.8. The GUI can also be styled with CSS under the `Advanced Composite View Layout` tab in Figure 7.8. When there are multiple components consisting of input widgets collected in one component, the generated GUI combines the GUIs of the sub-components together to create one GUI consisting of all the specified input parameters. The full GUI is shown in Figure 7.9. The model parameters are not part of the main KNIME GUI, as they are separate GUIs generated by their respective model nodes.

The full KNIME GUI is shown in Figure 7.9. The GUI uses more space for each parameter than the MATLAB GUI does, and it has a different structure. Some of the input field changes are due to the automatic generation of the GUI, as the widget nodes provides less possibilities for customization than developing a GUI in MATLAB. As an example, there is no way to create input fields that depend on other fields, which causes `Select depth` to be split into two input fields, `Select depth (If selected ECCO)` and `Select depth (If selected Ocean Data)`. The original GUI had one depth input field with values based on the database selection. The main change to

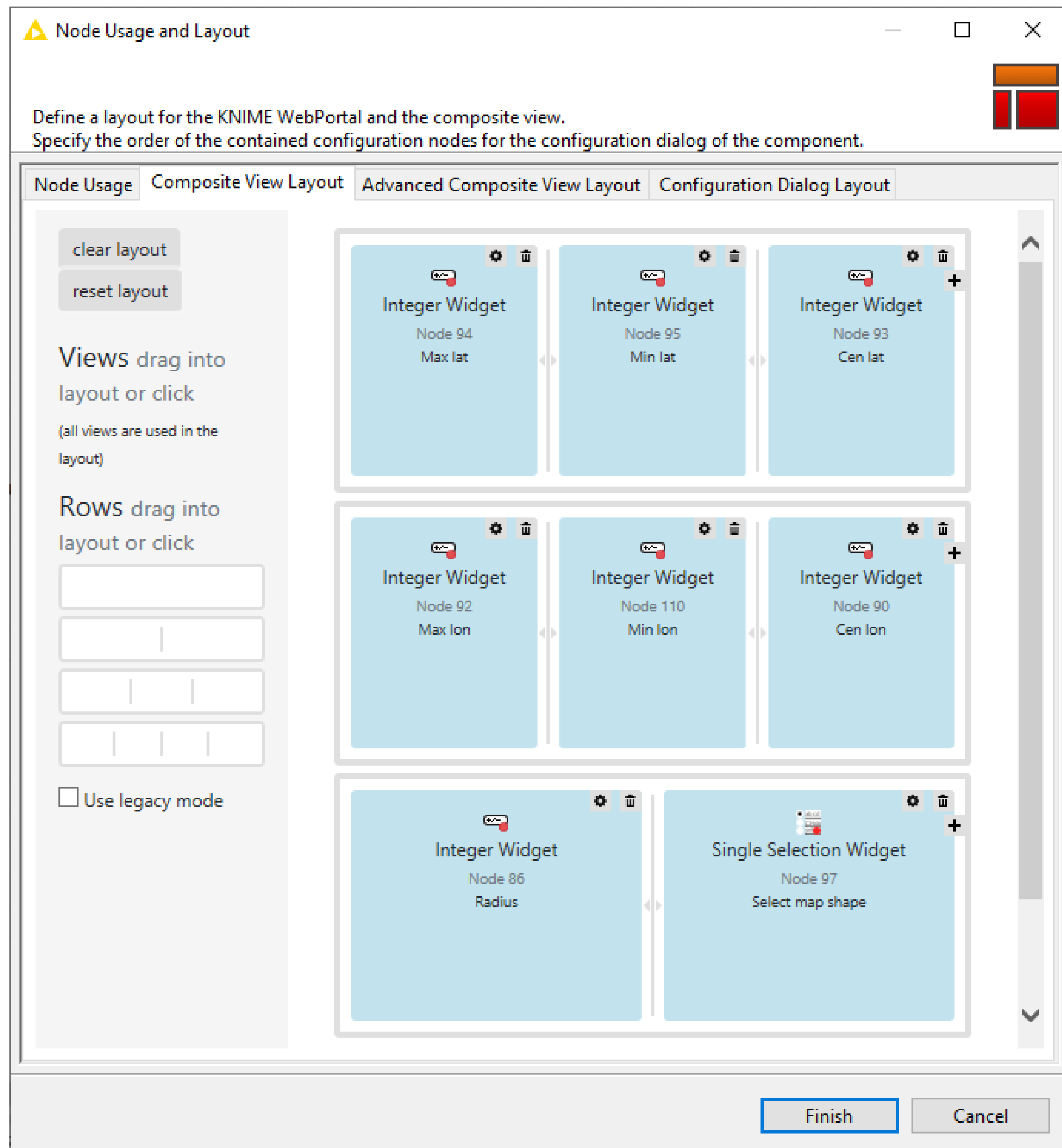


Figure 7.8: The KNIME view to configure how the generated GUI is organized. The blue boxes represent the parameters, and where they are placed relative to each other in the GUI. For instance, the `Max latitude` parameter is located above the `Max longitude` parameter, as can be verified in Figure 7.6.

Take input

Select database: IBCAO (2min)
Select depth (if selected ECCO): 5
Select depth (if selected Ocean Data): 6.7

Select coastline resolution: Low resolution coastline

Select source file: [illegible]\knime-workspace\ArcticOcean\ArcticOceanFiles\priorRuns\sources_OAATEX.dat

Select receiver file: [illegible]\knime-workspace\ArcticOcean\ArcticOceanFiles\priorRuns\receivers_OAATEX.dat

Max latitude [°N]: 90
Min latitude [°N]: 70
Cen latitude [°N]: 85

Max longitude [°E]: 180
Min longitude [°E]: -180
Center longitude [°E]: -5

Radius: 15
Select the shape of the map: rectangular

Selected receiver index: 2
Selected source index: 1

DR[km]:C: 10
DR[km]:Z: 5

Profile type: Sound speed c(r,z)
Sound speed database: ECCOV4
Time variation (used with WOA): Annual

Select model(s) to run: RAM, MPIRAM, Bellhop, Eigenray

Save model?

Filename to save file (empty for timestamp)

Reset Apply Close

Figure 7.9: The full GUI generated by KNIME from widget nodes. The map section from Figure 7.6 is the middle section of the full GUI, starting from `Max latitude [°N]` to `Select the shape of the map`.

the generated GUI compared to the original GUI, is that there is a new visual layout. The input field names, spacing, fonts, and colors are different. Another change is that the model input parameters are in a separate context. The original GUI creates a pop-up window that asks for the input for the model, while the KNIME workflow uses the model nodes to generate GUIs to take the model specific parameters.

7.3 React JSON Schema Form

Apache Airflow does not have built-in functionality to generate a user interface, which means that there is need for a tool that can generate a GUI. RJSF [83] is a React component that can build HTML forms based on a JSON schema where the input parameters are defined. RJSF is well-suited to generate a GUI for the Airflow workflow, as it uses JSON to generate a form and supports multiple different input types, such as string, number, integer, boolean, arrays, selections (one, one or more, all), and file selection. To generate the GUI, a JSON schema is created in the React application and the library generates a form from it. The JSON schema can also be sectioned into multiple objects with separate headlines, making the GUI simple to navigate. Another useful feature is that dependencies between input fields can be added, which allows the developer to add different options for an input field, depending on selections made by the user in another. RJSF also allows to define which fields are required. The complete GUI is too large for one figure, but is split up into Figure 7.10, Figure 7.11, and Figure 7.12, which together show the GUI in its entirety.

In RJSF, it is possible to configure the visual layout of the GUI. RJSF supports different UI-themes, such as material-ui and bootstrap-4. Additionally, configurations for specific input fields can be added as a separate schema to control how the input field is shown, i.e. to show radio buttons instead of a drop-down menu. Figure 7.10 shows how a RJSF generated GUI with the material-ui theme looks. The GUI shown in the figure has multiple drop-down fields, for instance `Coastline resolution`, which is defined as an enum in the schema. It also has integer fields for the latitude and longitude input fields, and file selections for source and receiver files.

The main changes to the GUI compared to the MATLAB GUI, is that it has a new visual layout, and that all user input is in one window. The new visual layout includes colors, style, naming, spacing, and ordering of parameters. For the visual layout, the form adds every input field to a separate line underneath each other, as shown in the figures. It is possible to create a grid layout for the form, to make the GUI more compact, but that would require writing custom code, which defeats the purpose of an automatically generated GUI. The advantage of having all the input fields in one place and not spread across different nodes or windows, is that the user only has to interact with one window.

Arctic Ocean GUI

Input the parameters to run the Arctic Ocean acoustic models

Create map

Coastline resolution *
High

Coloring database *

Minimum longitude [°E] *
-180

Maximum longitude [°E] *
180

Center longitude [°E] *
-5

Minimum latitude [°N] *
70

Maximum latitude [°N] *
90

Center latitude [°N] *
85

Map radius *
15

Map shape *
Rectangular

Browse... No file selected.

Select file to load sources from

Browse... No file selected.

Select file to load receivers from

Figure 7.10: The top part of the GUI generated by RJSF for the Airflow workflow. The shown section takes the user input required for the map, in addition to source and receiver information.

Model parameters

Select which models to run and cofigure the model runs.

Select source *

1

Input source index (1 is the first line in the input file). Use the map for a visual representation of where the sources and receivers are.

Select receiver *

3

Input receiver index (1 is the first line in the input file). Use the map for a visual representation of where the sources and receivers are.

DR[km]:C *

5

Sound speed step size

DR[km]:z *

10

Bathymetry step size

Sound speed database *

ECCOv4

Sound speed database for modeling

Select timestep *

Annual

Select when to run the model for. ECCO only supports Annual.

Select profile *

Sound speed c(r,z)

Select profile for modeling

Figure 7.11: The middle part of the GUI generated by RJSF for the Airflow workflow. The shown section takes the common parameters that all the acoustic models require for modeling.

Select model(s) to run

Select one or more models to run with the given parameters.

- [x] RAM
- [ ] MPIRAM
- [x] Bellhop
- [ ] Eigenray

RAM

frequency *

100

Bellhop

Frequency *

50

Simulation type *

- [ ] Raytracing
- [x] Eigenrays
- [ ] Incoherent transmission loss
- [x] Semi-coherent transmission loss
- [ ] Coherent transmission loss

SUBMIT

Workflow not submitted

Click here to se the workflow run status

Figure 7.12: The bottom part of the GUI generated by RJSF for the Airflow workflow. The shown section takes model input parameters. In this example, the RAM and Bellhop acoustic models are selected, and their specific parameters are set. The parameters for the other models are hidden.

The JSON schema used to create the GUI contains information about title, description, type (object, string, integer, number, boolean), properties, dependencies, and required properties. The section `Create map` in Figure 7.10 contains an object that consists of properties. The properties can either be new objects or a single input field. Objects also contain information about which of its properties are required through the `required` property.

Listing 7.1 contains a piece of the JSON schema for the GUI in Figure 7.10. Only the schema for `minlon`, `maxlon`, `radius`, `shape`, and `source_file` input fields within the model object is shown, as the rest of the schema shows the same concepts. In line 4, the required fields within the model object are set, which are all of the input parameters for the map. The `minlat`, `minlon`, and `radius` parameters are integer input fields. They all have titles and default values set. The title is the name of the input field shown to the user. The default values are the values shown in the GUI before the user interacts with it. The value -180 is the default for `minlon`, 70 is the default for `minlat`, and 15 is the default for `radius`. The `shape` parameter is an enum of strings, and one item in the enum can be selected. The string type indicates that the returned value from `shape` is a string. By default, enums show up as drop down menus. The `enumNames` property indicates how the enum options are shown to the user in the GUI. The `enum` property indicates how the values are sent when submitting the form. This is useful for the Arctic Package, as the old package frequently used integer values instead of names to identify parameters such as sound speed database.

Another useful property of RJSF is the `dependencies` property shown in Listing 7.2. This property is used when selecting which models to run, and makes it possible to only show the input fields for the models the user wants to run. Figure 7.12 shows how the GUI looks when the user selects to run the RAM and Bellhop models. The input fields related to RAM and Bellhop are visible, but the input fields for MPIRAM and Eigenray are hidden. The dependency property works by specifying options for a parameter. For the RAM model, the options are either that `run_ram` is false, in which case there are no input fields for RAM shown, or that `run_ram` is true, in which case the input field `frequency` is shown to the user. The `dependencies` parameter specifies which property the dependencies control, in this case `run_ram` (line 2). The `oneOf` property specifies that the schema is valid if exactly one of the sub-schemas are valid. Lines 4-10 covers the case where `run_ram` is false, while lines 11-26 covers the case where `run_ram` is true.

When the user has provided all of the input to the GUI, the user clicks the purple `SUBMIT` button in Figure 7.12 to trigger the workflow. The React application then sends a POST request containing the user input as payload to the Airflow REST API. The request triggers a DAG execution with the user input.

To avoid issues with cross-origin resource sharing (CORS) when sending requests from the React application running at `localhost:3000` to the Airflow REST API running at `localhost:8080`, a proxy to `localhost:8080` is set in the `package.json` file in the React application. A better solution would be to set the access control headers for the REST API, but no mechanism was found to achieve that in the

```
1     map: {
2       title: "Create map",
3       type: "object",
4       required: ["minlon", "maxlon", "minlat", "maxlat", "cenlon",
        ↪ "cenlat", "source_file", "receiver_file", "radius",
        ↪ "shape"],
5       properties: {
6       (...)
7         minlon: {
8           title: "Minimum longitude [°E]",
9           type: "integer",
10          default: -180,
11        },
12        minlat: {
13          title: "Minimum latitude [°N]",
14          type: "integer",
15          default: 70,
16        },
17        (...)
18        radius: {
19          title: "Map radius",
20          type: "integer",
21          default: 15,
22        },
23        shape: {
24          title: "Map shape",
25          type: "string",
26          default: "rectangular",
27          enum: ["circular", "rectangular"],
28          enumNames: ["Circular", "Rectangular"],
29        },
30        source_file: {
31          title: "Select file to load sources from",
32          description: "Select file to load sources from",
33          type: "string",
34          format: "data-url",
35        },
36        (...)
37      },
38    },
```

Listing 7.1: Parts of JSON schema describing the input parameters to generate the GUI. The `(...)` sections indicate that some of the code was left out for brevity.

```
dependencies: {
    run_ram: {
        oneOf: [
            {
                properties: {
                    run_ram: {
                        const: false
                    }
                }
            },
            {
                properties: {
                    run_ram: {
                        const: true
                    },
                    RAM: {
                        required: ["freq"],
                        properties: {
                            freq: {
                                title: "frequency",
                                type: "number",
                                default: 100,
                            }
                        }
                    }
                }
            }
        ]
    }
}
```

Listing 7.2: The JSON schema for taking input for the RAM model. The dependencies specify when the input fields for RAM show up. If the `run_ram` parameter is false, nothing is showed, but if it is true, then all input fields specified in for RAM are shown.

current version of Airflow. There is an open issue about it from 2019 [84].

The request code is shown in Listing 7.3, and is triggered at the `onSubmit` event of RJSF form. The request is a POST request (line 3), as it needs to trigger a DAG execution. At line 9, the payload is set to be the form data from the RJSF GUI. The form data is the JSON representation of the user input. For the parameters from Listing 7.1, the `event.formData` would be as shown in Listing 7.4, assuming the user did not change the default values. When the trigger request is sent, the workflow progress is shown at the Airflow webserver. It is possible to fetch and display workflow status in the GUI as well, this was not a priority, as the information

is easily accessible through the Airflow webserver.

```
const runDag = (event) => {
  return fetch(api_url + 'dags/' + dag_id + '/dagRuns', {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      conf: event.formData
    })
  })
    .then(res => res.json())
    .then((json) => {
      setWorkflowStatus("Workflow with id: " + json.dag_run_id
       ↪ + " submitted");
    })
    .catch((error) => {
      console.log(error)
    });
};
```

Listing 7.3: JavaScript code implementing the sending of a request with parameters from the GUI application to the Airflow REST API.

```
{
    "map": {
        "minlon": 180,
        "minlat": 70,
        (...)
        "radius": 15,
        "shape": "rectangular",
        "source_file":
         ↪ "data:text/plain;name=<filename>;base64,<encoding>",
        (...)
    }
}
```

Listing 7.4: The outputted format of the user input to the Airflow workflow. The data can be used without any changes as Airflow dag configuration. (...) indicates skipped values.

Chapter 8

Evaluation

To evaluate the developed workflows and the software refactoring, some evaluation criteria must be established. There exists several models to evaluate software quality, such as the ISO/IEC 25010 standard [85], McCall's software quality model [86], Dromey's software quality model [87], and the FURPS software quality model [88]. Each of the models specify quality criteria. The quality criteria for each model are summarized and compared in Table 8.1. Some criteria are common across models, while some are only considered by a single model. Note that each of the McCall, Dromey, and FURPS models were developed in 1977, 1996, and 1992 respectively, while the ISO/IEC standard was released in 2011 and last reviewed in 2017.

Table 8.1: Evaluation criteria overview for the software quality models. The ISO/IEC criterion name functional suitability was shortened to functionality and performance efficiency was shortened to performance.

Criteria	ISO/IEC 25010	McCall	Dromey	FURPS
Functionality	✓		✓	
Performance	✓			✓
Compatibility	✓			
Usability	✓		✓	✓
Reliability	✓	✓	✓	✓
Security	✓			
Maintainability	✓	✓	✓	
Portabilty	✓	✓	✓	
Correctness		✓		
Efficiency		✓	✓	
Integrity		✓		
Flexibility		✓		
Interoperability		✓		
Reusability			✓	
Supportability				✓

Table 8.1 shows that reliability is part of all four models. Usability, maintainability, and portability is part of three models. The criteria functionality, performance, and efficiency are part of two models. Some of the criteria listed as a top level criterion in a model are also sub-criteria of other models, such as correctness, which is a

sub-criteria of functionality in the ISO/IEC 25010 standard.

Ozakaya et al. [89] used data with 1,072 scenarios to find trends in quality attributes looking for similarities and differences with existing methods for specifying quality attributes. They found the distribution percentage for the scenarios, by taking the number of times each criteria was expressed as a top-level quality attribute, divided by the total number of scenarios. Modifiability, performance, usability, maintainability, and interoperability are the top five on the list with respectively 14.1%, 13.6%, 11.4%, 8.5%, and 7.8% mentions. Reliability comes in 9th with 5.7% and portability 17th with 0.8%. Functionality is not on the list.

Taking into account the data in Ozakaya et al., the reoccurring criteria across different models, and the age of the models, the ISO/IEC 25010 standard was selected to evaluate the quality of the Arctic Package. Section 8.1 describes the criteria from the ISO/IEC 25010 standard in detail, and discuss their relevance in the context of the Arctic Package. Section 8.2 describes how the evaluation is performed, and the subsequent sections from Section 8.3 to Section 8.9 discuss the original Arctic Package, the KNIME workflow implementation, and the Airflow workflow implementation with regards to each of the evaluation criteria. Section 8.10 summarizes the evaluation.

8.1 Criteria for Evaluation of the Arctic Package

To evaluate the Arctic Package, some appropriate evaluation criteria must be established. Not all criteria described in the ISO/IEC 25010 standard are useful to apply to the Arctic Package. For example, security is not relevant as the application is running locally and is not exposed to others than the researchers at NERSC. In addition, there is no sensitive data being handled. It might be relevant in the future, if the Arctic Package is hosted and made available to a larger user base.

The criteria from the ISO/IEC 25010 standard are functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability [90], [91]. Of those, all but security are considered.

8.1.1 Reliability

Reliability is part of all four software quality models. ISO/IEC 25010 defines it as how well the system can perform its functions under different conditions at a specified period of time. They define the sub-characteristics: maturity, availability, fault tolerance, and recoverability. Maturity is how well the system meet the requirements for reliability under normal operation. Availability is the fraction of time that the system is operational and accessible. Fault tolerance is how well the system handles errors, and whether or not it is still operational when an error occurs.

Reliability is not crucial for the Arctic Package, as it is a software package installed locally that can be modified by the users that use it. It is not hosted online, nor does it have a large user base that depend on it. However, in order to make the package

more relevant and used, the Arctic Package needs to be reliable and produce reliable results.

8.1.2 Usability

Usability is concerned with the user interface quality. The ISO/IEC 25010 standard specifies the following sub-characteristics: appropriateness recognizability, learnability, operability, user error protection, user interface aesthetics, and accessibility.

Usability is important in the Arctic Package. The package had a complex user interface, and even after Van den Bergh and Klockmann made changes to simplify the user interface, there is room for improvement. As the package has not been used a lot, partly due to the complex user interface, it is important to try to address the issue of usability. However, measuring usability is challenging, as the criterion and sub-criteria are highly subjective.

8.1.3 Portability

Portability is concerned with how effective a system can be transferred from one environment to another. The ISO/IEC 25010 standard specifies the sub-characteristics of adaptability, installability, and replacability. Portability is important as the Arctic Package is used in multiple environments and each researchers computer is a separate environment. The package needs to be easy to set up and use for all environments, such as for different operating systems. Portability is also important in order to deploy the Arctic Package to a server later on.

8.1.4 Maintainability

Maintainability is how easy it is to modify the system to improve, correct, or adapt it. The ISO/IEC 25010 standard specifies the following sub-characteristics: modularity, reusability, analyzability, modifiability, and testability. The ISO/IEC standard specifies reusability as a sub-criterion of the maintainability criterion, while Dromey has reusability as a top level criterion.

Maintainability is an important criterion to keep the package up to date. If it is hard to update the acoustic models and libraries used, there is a higher chance that the outdated versions are used. When maintainability is difficult, it is also hard to include new features, such as new models, one of the wishes from NERSC from the requirements listed in Section 3.4.

8.1.5 Compatability

Compatability is how well the system can perform its functions while sharing resources with other systems, and how well it exchanges information with other systems. The sub-characteristics for compatibility is co-existence and interoperability. Co-existence is how well the system can co-exist with other systems and how it impacts other programs. Interoperability is how well it can exchange information with other systems and how well the information can be used.

Co-existence is important for the Arctic Package, as the Arctic Package is running locally on the computer and the user most likely has other applications in use as well. Co-existence is especially important in the revised version of the Arctic Package, as all the input is given prior to running the workflow. This means that the user does not need to interact with the Arctic Package again until it has finished running, and can perform other tasks while the workflow runs. Interoperability is important for the Arctic Package to operate correctly, as it needs to interact with the acoustic models and, in the Airflow workflow, the GUI.

8.1.6 Functional Suitability

ISO/IEC 25010 explains functional suitability as how well a system provides the functions the stakeholder needs. To decide how well the functional suitability criterion is fulfilled, three sub-characteristics are defined: functional completeness, functional correctness, and functional appropriateness. Functional completeness is how well the functions cover the the specified tasks, functional correctness is whether or not the results from the system are correct and accurate, and functional appropriateness is how well the functions fulfill their task.

Functionality is important in the Arctic Package to make sure it is relevant and effective. Some of the functionality is added to ease usability, such as running the acoustic models for multiple timesteps in one model run. Functionality is also important as the Arctic Package has some functionality that is already in use, and that functionality should be a part of the revised package. The ISO/IEC 25010 functional suitability criteria, also spans over the correctness criteria from McCall's model, as functional correctness is a sub-criteria of the ISO/IEC 25010 standard.

8.1.7 Performance Efficiency

Performance efficiency measures how well the system performs and the amount of resources that are used by the system. ISO/IEC 25010 defines three sub-characteristics: time behaviour, resource utilization, and capacity. Time behaviour measures response, processing times, throughput rates, and whether they meet the requirements. Resource utilization measures how much of different types of resources that is used by a system during execution. Capacity is related to the limits of the system, and whether the maximum capacity is sufficient.

For the Arctic Package, some of the performance will depend on the models and their implementation, but the setup of the Arctic Package and processing of in- and output data also has an impact.

8.2 Software Quality Evaluation

The software quality is evaluated for the original Arctic Package, the workflow implementation in KNIME, and the workflow implementation in Airflow. Each of the criteria functional suitability, performance efficiency, compatibility, usability, reliability, maintainability, and portability are evaluated, and a score from 1 to 5 is given based on an assessment of how well they satisfy the criteria. 1 is the lowest

score and 5 represents the best score. The scale is based on the needs for the Arctic Package, and is not meant to be comparable or transferable to other systems.

8.3 Reliability

Reliability has the sub-characteristics of maturity, availability, fault tolerance, and recoverability. Availability is not the most critical criterion in the Arctic Package, as it is started when needed and can be restarted by the user after a crash. This means that it is available as long as the user has installed the software. However, having unreliable software with frequent fault occurrences can contribute to less usage of the package and also unreliable results, therefore the reliability is evaluated for the package.

There were no unit-tests in the original MATLAB implementation, and no tests were added for the workflow implementation of the Arctic Package. Tests could have been created to increase the reliability and detect errors in the software, but implementing the core functionality was prioritized.

8.3.1 Original MATLAB Implementation

In the original implementation of the Arctic Package, failures mostly happens due to user errors. These are discussed in Section 8.4. However, there are also bugs in the code that are unavoidable, even with correct use. The two main bugs known are:

- GECCO2007 and Ocean Model sound speed database produce errors.
- Depths for Ocean Model that were used in computations were different from the depths presented in the GUI.

With regards to fault tolerance, the errors are shown in the MATLAB console, but the program does not crash. The user can retry the modeling using different parameters without restarting the program.

In total, the original MATLAB implementation is for the most part reliable. There are few crashes and hangs. Most of the errors that occur are due to user errors. However, there are also the two bugs, where especially the sound speed database bug affects reliability, as two of the four database choices produce errors. This causes the overall reliability score for the MATLAB implementation to be 3.

8.3.2 KNIME Workflow

In the KNIME workflow, the depth bug from the original Arctic Package is resolved by removing the options of modeling with GECCO2007 and Ocean Model. This decreases the fault rate for the workflow. However, adding KNIME to the Arctic Package, adds another layer of software, which increases the possibility for faults to occur. KNIME sometimes experiences the errors in Listing 8.1 and Listing 8.2. The error in Listing 8.1 is caused by the library `m_map` that uses the `evalin` MATLAB function. The same error also occurs when using the plot files for Bellhop. The error in Listing 8.2 seems to be caused by running out of memory, but this needs to be

further investigated to know for certain.

```
1 KNIME: execution error:
2 Error using evalin
3 Unrecognized function or variable 'snipped_3679691901025884322'.
```

Listing 8.1: MATLAB `evalin` error that occurs when running the KNIME workflow.

```
1 Execute failed: Object attempting to be received cannot be
  ↪ transferred between Java Virtual Machines
```

Listing 8.2: MATLAB error about Java Virtual Machines that occurs when running the KNIME workflow.

The errors in KNIME seems to be an issue with the integration of MATLAB and KNIME. The issues do not occur in Airflow, nor when the MATLAB functions are run outside KNIME. The errors are much more frequent when the computer that is used to run the KNIME workflow is also used to perform other tasks, such as browsing the web. The issue can be caused by running out of memory, as KNIME, the Docker containers for the acoustic models, and MATLAB consumes a lot of memory. The computer used in the evaluation has 8GB RAM, and the frequency of errors is likely to go down in a computer with more memory available.

In addition to the errors, the MATLAB scripting nodes produces the warning in Listing 8.3, which indicates that if the scripting nodes are not updated, they will not be compatible with newer versions of MATLAB.

```
1 Warning: A value of class "struct" was indexed with no subscripts
  ↪ specified. Currently the result of this operation is the
  ↪ indexed value itself, but in a
2 future release, it will be an error.
```

Listing 8.3: MATLAB warning caused by the MATLAB Snippet nodes.

With regards to fault tolerance, KNIME continues the workflow execution for the branches that does not depend on the failing task. In addition, only the data in the executing node is lost. All data from other nodes that executed are saved. The only node that needs to be re-run to resolve the error is the one with an error and subsequent nodes that depend on it. This makes errors easier to handle and debug, as the node can be debugged in the exact context it failed.

Overall the experience with regards to reliability in KNIME, is that it can be unreliable, but that errors can be easily resolved. Usually when errors first occurred, they would continue also when nodes were run again. However, most of the time the solution was to restart KNIME or MATLAB, and occasionally it was necessary to restart the computer. Errors would often occur when the workflow was run frequently in a short period of time and when the computer used resources for other programs. The unreliability of KNIME is also evident from variation in the execution times measured. The difference in execution time was up to two minutes between the slowest and fastest run when the same set of parameters was used. This is almost double the time of the fastest execution time measured in KNIME. This is discussed further in Section 8.9.

A simple test was performed to test reliability on the completed workflow. The test consisted of running the workflow 15 times in a row and count errors and crashes. No other programs were running on the computer and the test was performed after a restart of the computer. For the 15 executions, the workflow encountered the error in Listing 8.2 once and froze once. Which gives a 2/15 failure rate for the test. In total, also considering the performance variance and overall experience, that gives the KNIME workflow a reliability score of 2.

8.3.3 Airflow Workflow

In the Airflow workflow as in the KNIME workflow, the depth bug from the original Arctic Package is resolved by removing the options of modeling with GECCO2007 and Ocean Model. On the other hand, there are more elements that can go wrong in the Airflow workflow compared to the original Arctic Package. The workflow and the GUI application must both function correctly in addition to the MATLAB source code. With Airflow, the errors and warnings seen in KNIME does not occur. If an error does occur, it does not affect the workflow when starting a new run through of the workflow.

A drawback with the error handling in Airflow, is that if the workflow fails, the task that failed cannot be re-run without running the entire workflow. However, Airflow can be configured to retry tasks if an error occurs, which can help the workflow mitigate errors that happen during a workflow execution. A failing task does not affect tasks in other branches of the workflow, which means that even if a task fails, the tasks that do not depend on it can complete.

Overall the experience with regards to reliability in Airflow is good. Errors rarely occur and no crashes were encountered. The computer did experience some freezes when running the workflow, especially when allowing the workflow tasks to run in parallel. The same test as for the KNIME workflow was performed to test reliability on the finished Airflow workflow. The tasks in Airflow were not set to retry on fail. Over the 15 workflow executions one error occurred. The error is shown in Listing 8.4. Unlike in KNIME, the error only occurred once and no restart of the software was required. The test was also the first time the error appeared. This and the overall experience while working with Airflow gives the Airflow workflow a total reliability score of 4.

```
1     Inconsistency detected by ld.so: ../elf/dl-tls.c: 481:
      ↪  _dl_allocate_tls_init: Assertion `listp->slotinfo[cnt].gen <=
      ↪  GL(dl_tls_generation)' failed!
```

Listing 8.4: The Airflow error that occurred in one of the tasks. The error might be caused by a glibc bug [92].

8.4 Usability

Usability is mostly a subjective measure, and it is hard to find metrics for measuring usability. Due to the Covid-19 situation and limited availability of relevant people to test the usability, it is not possible to conduct a comprehensive investigation to get reliable results. Instead it is possible to investigate specific changes to the GUI and estimate how they affect the usability.

The sub-characteristics for usability in the ISO/IEC 25010 standard is appropriateness recognizability, learnability, operability, user error protection, user interface aesthetics, and accessibility. Appropriateness recognizability is how easy it is for a user to recognize if the system is appropriate for their needs. For this project, it is assumed that the user of the Arctic Package decides to use the package based on prior knowledge that it does fit their needs, thus the appropriateness recognizability sub-characteristic is not considered.

The original GUI had some usability issues:

- It was hard to tell which window to interact with, e.g. when to use the map, the console, and the main GUI.
- Errors from input and missing user input was shown in the MATLAB console, where the user might not see it.
- A bug where the depths shown for the Ocean Model are not the depths actually used.
- It was easy to get user errors when interacting with the map, because the positions selected on the map can be ambiguous and the script might select the wrong source or receiver.

These issues causes the usability score for the original MATLAB implementation to be 1. A positive note on the usability, is that the interactive map is easy to work with, and gives a good overview of the area, sources, and receivers available.

8.4.1 Usability Changes Common in Airflow and KNIME

This sections discusses the common changes that affect usability for the KNIME workflow and Airflow workflow. The main changes are that:

- The user interactions with the MATLAB console were removed.

- The interactive map was removed.
- The parameters that were rarely used and unused ones were removed.
- Pop up menus were removed.

Table 8.2 shows how the changes have affected usability of the Arctic Package. The rest of this section discusses the changes and how they influence each of the usability criteria.

Table 8.2: Usability changes compared to the original GUI of the Arctic Package. A plus indicates an improvement for the criterion, while minus indicates that it has worsened.

Change	**Learnability**	**User error protection**	**Aesthetics**	**Accessibility**	**Operability**
Removing console interaction	+			+	+
Removing interactive map	÷	+		+	÷
Removing unused parameters	+	+			
Multiple model runs					

In the original version of the Arctic Package, the user occasionally had to interact with the MATLAB console instead of the GUI or map. An example of this is when the `GET PREDICT` button in the GUI (Figure 7.1, Section 7.1) is clicked and a message `Click near the Desired Source` appears in the console, but nowhere in the GUI. The same issue occurs when the `Clear` button is clicked. Another way the console was used, is to show error messages, and as there are no checks for input errors or missing input by the GUI, the MATLAB console messages occur frequently.

Removing the interactions with the MATLAB console has the benefit of removing uncertainty about where and how to interact with the Arctic Package. That in turn increases learnability and operability, as there are fewer mechanisms to learn and operate. The workflow versions contain one input view, as opposed to an input view, a map, and the MATLAB console. It also increases accessibility by showing error messages to the user through the GUI instead of through the MATLAB console.

Another change is that the interactive map from the original Arctic Package was replaced with a map that cannot be interacted with, but can be used to view the area, sources, and receivers. An advantage of removing the interactive map, is that the user only has to input information to the GUI once. Previously the user had

to go through a sequence of inputs and button clicks to create the map and then perform the modeling. A typical sequence of interactions would be:

1. Input map parameters, click `Set Map`, and wait for plotting.
2. Select source file, click `Get`, and wait for plotting.
3. Select receiver file, click `Get`, and wait for plotting.
4. Click `Plot paths` and wait for plotting.
5. Select an acoustic model and click `GET PREDICT`.
6. Input model parameters for the selected model in the new window, click `GET PREDICT`, and wait for model results.

It is not possible to run the models without first plotting the map, the sources, and the receivers.

In the new approach, the user inputs all the information at once and then starts the workflow. This increases usability by reducing user interactions and wait times between interactions with the GUI. A drawback is that the map is not visible to the user when the acoustic model parameters are set. A workaround to show the map, is to run the workflow once without running any of the models, and use the resulting map to select input values for the acoustic modeling. However, removing the map decreases learnability and operability, as it is not an intuitive solution.

An improvement from removing the interactive map, is that any ambiguity about which source or receiver was clicked is removed. The original version would register a click on the map and try to find the closest source or receiver. The new map is shown in Figure 8.1, and contains numbered sources and receivers. If the user wants to model from source 4 to receiver 8 there is no uncertainty about which source and receiver the user wants. However, if the user clicked somewhere between receiver 6 and 7 in the original version of the Arctic Package, either one of them could be selected by the package. The user does not know for sure which. The new map increases usability, specifically the user error protection sub-characteristic.

The advanced settings view (Figure 7.4, Section 7.2) is removed in the new GUI. The advanced settings are useful for debugging the package, but are generally not much used for regular model runs. This is one of the reasons Van den Bergh and Klockmann extracted the functionality to an advanced window [19], [20]. The selections in the advanced window can still be performed by adding new sources and receivers to the source and receiver files, which are given as input to the workflow. Removing the advanced option gives a minimal GUI with less information to process, which makes it easier to use. A drawback is that in order to make the changes that previously had their own menu, the user needs to change the source and receiver files, which is inconvenient compared to configuring them through the GUI.

In addition to removing the advanced menu, some of the less used or not used parameters were removed. The `Array #` parameter in Figure 7.1 was not fully implemented in the Arctic Package and its intended use is unclear. Another parameter that

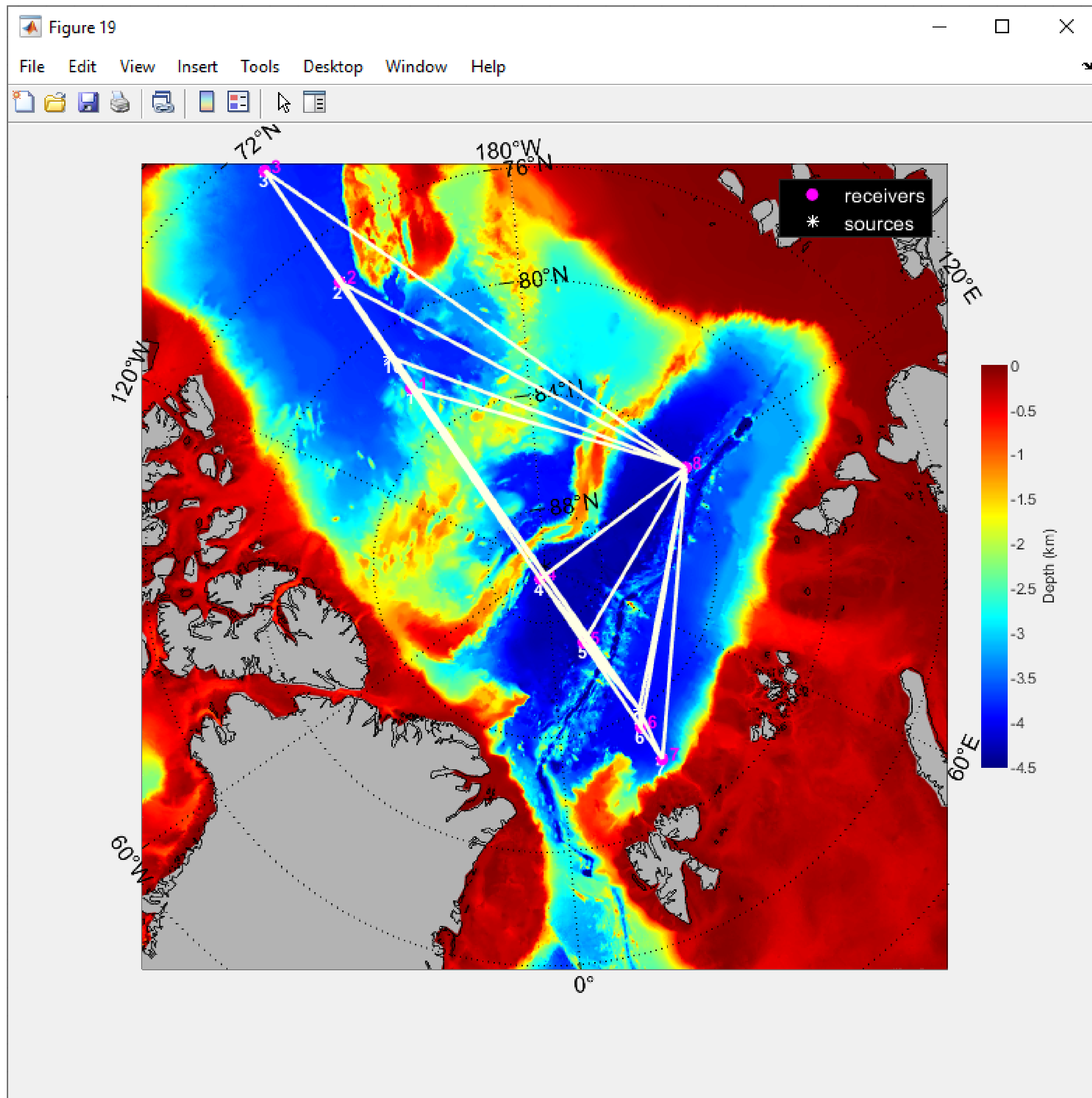


Figure 8.1: The map plot in the new version of the Arctic Package. The white numbers identifies the sources and the magenta number identifies the receivers. A legend is also added to the plot to identify which points that are sources and which are receivers.

was in the GUI, but is not used is the `Rcvr Dpt [m]` parameter. For that reason the parameter and the supporting MATLAB code was removed. Two of the other parameters `Timespan (min,max) [s]` and `Depthspan (min, max) [-km]`, that set the axes limits in the results, were also removed. Removing them reduces user errors, as the parameters can be set to values that does not make sense, such as setting min/max to the same values or to set min to a higher value than max. In addition, it is hard to set the axes to values that fit the results, as the results are unknown at the time of modeling. It is simpler and more robust to let the Arctic Package decide the time- and depthspans based on the results, which is also what happens when the values are set to `-1, -1`.

Removing the unused parameters increases learnability because there are fewer param-

eters to learn. In addition, the specific parameters removed (`Timespan (min,max) [s]` and `Depthspan (min, max) [-km]`) are parameters that are hard to set to appropriate values, which further increases learnability and also user error protection.

8.4.2 KNIME GUI

The KNIME GUI for the Arctic Package is shown in Figure 8.2. It takes up more space for each parameter than the MATLAB GUI does, and it has a different structure. Some of the input field changes are necessary because KNIME automatically generates the GUI from widget nodes, and there are less possibilities for customization than when writing MATLAB code. The changes in the KNIME GUI compared to the MATLAB GUI are:

- New visual layout, input field naming, spacing, and input field positions are changed.
- Model input parameters are in a separate context. The original GUI creates a pop-up window that asks for the input for the model, while the KNIME GUI has separate input nodes and views for that.

Table 8.3 shows how the changes affected usability of the Arctic Package. The rest of this section describes why the changes affects the usability positively or negatively.

Table 8.3: Usability changes for the KNIME GUI compared to the original Arctic Package GUI. A plus indicates an improvement for the criterion, while minus indicates that it has worsened.

Change	**Learnability**	**User error protection**	**Aesthetics**	**Accessibility**	**Operability**
New layout	+			÷	+
Separate model input	÷				÷

The KNIME layout has reduced accessibility compared to the MATLAB GUI. Keyboard navigation is difficult, as the only indication of focus is a slightly darker colored line around the focused input field. In addition, drop-down selections do not close when navigating away using the keyboard. These issues are not present when using the cursor to navigate. In the original GUI there are also some keyboard navigation issues where drop-downs do not open (but can be navigated using arrow-keys) and the tabbing order is backwards between the buttons `Visualization of saved results` and `Plot paths`.

The new layout allows more space for each input parameter, which makes all the information and default values in the input fields visible. In the original GUI some of the text values were cut off and not visible until the user clicked the input field.

Take input

Select database: IBCAO (2min)
Select depth (if selected ECCO): 5
Select depth (if selected Ocean Data): 6.7
Select coastline resolution: Low resolution coastline
Select source file: ...\knime-workspace\ArcticOcean\ArcticOceanFiles\priorRuns\sources_CAATEX.dat
Select receiver file: ...\knime-workspace\ArcticOcean\ArcticOceanFiles\priorRuns\receivers_CAATEX.dat
Max latitude [°N]: 90
Min latitude [°N]: 70
Cen latitude [°N]: 85
Max longitude [°E]: 180
Min longitude [°E]: -180
Center longitude [°E]: -5
Radius: 15
Select the shape of the map: rectangular
Selected receiver index: 2
Selected source index: 1
DR[km]:C: 10
DR[km]:Z: 5
Profile type: Sound speed c(r,z)
Sound speed database: ECCOV4
Time variation (used with WOA): Annual
Select model(s) to run: RAM MPIRAM Bellhop Eigenray
Save model?
Filename to save file (empty for timestamp)
Reset Apply Close

Figure 8.2: The GUI generated by KNIME. All the user input except for model specific parameters are set here. This figure is identical to Figure 7.9.

As a consequence of the new layout, accessibility is increased.

Some additional information about the parameters is added to the new GUI, such as when the input fields apply. The name of the profile types that can be selected are also shown in addition to their mathematical formula e.g. `Sound speed c(r,z)` instead of `c(r,z)`. Another addition is that the depth input field in Figure 8.2, specifies that it is used for ECCO or for Ocean Data. This increases operability and learnability, as better naming makes the input parameters easier to understand.

A disadvantage in the new GUI compared to the original one, is that the new GUI has one depth input field for ECCO and one for Ocean Data. In the original GUI the `Depth` input field was disabled for the models it did not apply to, which is a solution that causes less visual clutter. Another drawback with the KNIME GUI is that less source and receiver file information is available. Unlike in the original GUI, it does not say when the file was created or how many sources or receivers there are in a file. Instead it shows the complete file path, which most of the time is not necessary to know. An improvement worth to note is that the `Depth` input field had a bug where the incorrect depths were shown for Ocean Data, which is resolved in the new version.

The KNIME GUI addresses the issue of communicating errors to the user by showing error messages in the KNIME console, which is similar to using the MATLAB console from the original GUI. However, there are also error messages and output shown in the MATLAB console for the MATLAB nodes. KNIME has additional handling of missing input by always requiring default values when setting up the widget nodes. An issue with that, is that KNIME saves the default source and receiver file path as well, which changes between computers. Consequently, the path must be overridden to an existing path each time the workflow is set up on a new computer. If the file does not exist, KNIME produces an error and stops execution the first time the missing value is used. This side effect of default values causes decreased user error protection, as it is easy to forget to set the path and KNIME does not stop the user from proceeding to run the workflow.

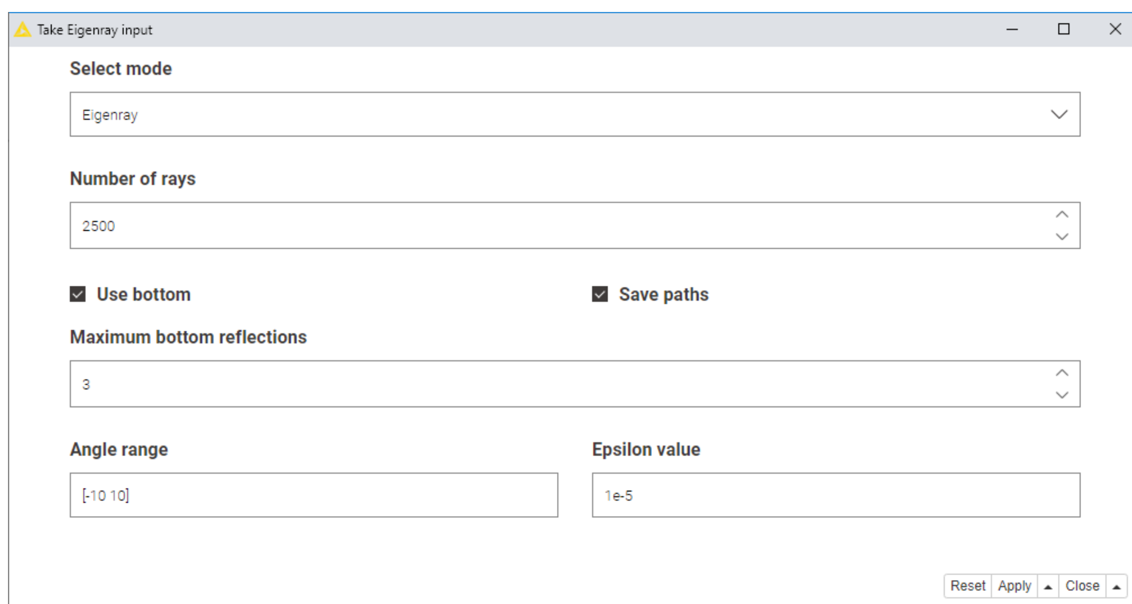


Figure 8.3: A model specific input view generated by KNIME. This figure is identical to Figure 7.3.

The KNIME workflow has separate GUIs for each of the acoustic models. An example of a model GUI in KNIME is the GUI for Eigenray in Figure 8.3. The GUIs are separate from the main GUI in Figure 8.2. This structure was chosen as it was not possible to hide the model specific parameters input field when a model was not

selected for a workflow run. Including the model parameters in the main GUI would mean that parameters not used when running the workflow would be visible, which can cause confusion about which parameters are used. The separation of the models parameters and the rest of the GUI causes operability and learnability to decrease. It is harder to operate a program with multiple different components, and the user has to learn how to work with all of the components.

In total, taking into account the common changes for both Airflow and KNIME and the KNIME specific changes, the usability is slightly improved. There are some usability improvements in learnability and operability, but those are overshadowed by the negative impact of having separate model input views and partly from removing the interactive map. The factor that improves the usability for the KNIME workflow, is the improvement user error protection. Overall, this gives the KNIME workflow a usability score of 2.

8.4.3 Airflow GUI

The Airflow GUI has the advantage of taking all input in one place, namely a form generated by RJSF from a JSON schema. The generated GUI gives each input field one line, and places the input fields underneath each other, as shown in Figure 8.4. It is possible to create a grid layout, but that would require writing custom code, which defeats the purpose of an automatically generated GUI. The Airflow GUI uses the Material-UI RJSF theme. The changes in the Airflow GUI compared to the MATLAB GUI are:

- New visual layout. Updated colors, style, input field naming, spacing, and layout.
- All the input is in one place, no multi-window approach or separate model views.

Table 8.4 shows how the changes affects usability in the Arctic Package with regards to the sub-factors of the usability criterion.

Table 8.4: Usability improvements for the Airflow workflow GUI compared to the original GUI. A plus indicates an improvement for the criterion, while minus indicates that it has worsened.

Change	**Learna-bility**	**User error pro-tection**	**Aesthetics**	**Accessi-bility**	**Opera-bility**
New layout	+	+	÷	+	+
One input view	+				+

The accessibility of the new GUI is improved. Mouse and keyboard navigation has no issues and the focus marking of the input fields is clear. Furthermore, the GUI shows error messages to the user when it receives input of the wrong type, such as

Arctic Ocean GUI

Input the parameters to run the Arctic Ocean acoustic models

Create map

Coastline resolution *
High

Coloring database *

Minimum longitude [°E] *
-180

Maximum longitude [°E] *
180

Center longitude [°E] *
-5

Minimum latitude [°N] *
70

Maximum latitude [°N] *
90

Center latitude [°N] *
85

Map radius *
15

Map shape *
Rectangular

Browse... No file selected.
Select file to load sources from

Browse... No file selected.
Select file to load receivers from

Figure 8.4: The top section of the Airflow GUI. The section shows the input fields related to the map. This figure is identical to Figure 7.10.

text values for a parameter that requires a number, and if any of the required input fields were not filled in. Another accessibility improvement, is that the GUI marks the required input fields with a star (∗), which makes it clear which input fields are required and not. The error handling also improves user error protection, as the user cannot input wrong values or start a workflow without filling in all input.

The new structure of the Airflow workflow shows each input field on one line, which causes the GUI to be quite large, Figure 8.4 only shows half of the GUI, even without showing the model input parameters. This decreases aesthetics of the GUI. The aesthetics is somewhat improved by focus marking using coloring and good sectioning of the GUI with headlines and sub-headlines.

The naming of the input parameters is improved in Airflow in the same way as in KNIME, with the addition of descriptions where they were found useful. That increases learnability and operability, as the user can refer to descriptions if they are uncertain.

The Airflow GUI also has the added benefit that it is possible to hide and show input fields based on the values of other input fields. This avoids the structure in KNIME where the `Depth` parameter is split into two input fields, depth for Ocean Model and depth for ECCO. The options for the `Depth` input field are also updated for the Ocean Model, which resolves the bug from the original GUI. This increases operability, as there are less visual clutter and all fields that are shown in the GUI are relevant for the modeling.

Figure 8.5 shows how the part of the Airflow GUI that takes input parameters for the acoustic models. The models RAM and Bellhop are selected to be run, and input fields for their parameters show up in the GUI. The other models input parameters are hidden. Having one input view has the same consequences as removing the interactive map and the interactions with the MATLAB console, as described in Section 8.4.1. Compared to the GUI in KNIME, the solutions in the Airflow GUI works better, because all of the user input is inputted in one place. The user does not have to navigate multiple different windows or configure different nodes to run the workflow. This increases operability, as it is quicker and easier to input all parameters in one place. It also increases learnability, as all information is collected in one place and there are no special mechanisms to remember.

In total, the usability has improved with the Airflow workflow, and the usability score for Airflow is 4. The drawback from removing the interactive map and from the large GUI, are small compared to the benefits, but enough to reduce the usability score from 5 to 4.

8.5 Portability

The three sub-characteristics of portability in the ISO/IEC 25010 standard are adaptability, installability, and replaceability. Portability is mainly affected by the new pipeline for the Arctic Package as described in Chapter 6. In this context, portability is concerned with how easy it is to adapt and install the software across several computers and operating systems.

The original MATLAB implementation did not include any instructions to set up the Arctic Package, which left the user to try figure it out by themselves. This meant looking at documentation and instructions for each of the acoustic models, and it might also be necessary to edit the commands that were used to run them in the MATLAB code. To run the acoustic models in the original Arctic Package, the models must be compiled and an executable created. The process of compiling the models should be simple, as each of them has a Makefile attached. However, the compilation had a tendency to fail. For example in the Bellhop model, the name of the folder containing the model was written as `bellhop` in the file structure, but in the Makefile it was set to be `Bellhop`. Additionally, some of the files would not compile. To compile the model, the Bellhop acoustic model source code was fetched from the original source and then compiled. All of the acoustic models required small adaptions to make them run on a new computer, either to the Makefiles or to the MATLAB scripts launching the acoustic models. The changes to the Makefiles were related to which Fortran compiler was used and the location of files and compilers in

Select model(s) to run

Select one or more models to run with the given parameters.

- [x] RAM
- [ ] MPIRAM
- [x] Bellhop
- [ ] Eigenray

RAM

frequency *

100

Bellhop

Frequency *

50

Simulation type *

- [ ] Raytracing
- [x] Eigenrays
- [ ] Incoherent transmission loss
- [x] Semi-coherent transmission loss
- [ ] Coherent transmission loss

SUBMIT

Workflow not submitted

Click here to se the workflow run status

Figure 8.5: The bottom section of the Airflow GUI. This section has checkboxes to select which acoustic models to run, and input fields specific for the acoustic models selected. This figure is identical to Figure 7.12.

the local file system.

Due to the difficulties described above, both installability and adaptability was poor with the original package. Installation could have been easier if there were

installation instructions attached to the Arctic Package. Adaptability was also not ideal, as each of the models seem to have computer specific run commands. For example the Windows command to run RAM is:

```
1 ! C:\cygwin64\bin\bash.exe --login -c "cd /cygdrive/c/Arctic.Ocean/
  ↪ && src/Collins_package/ram.exe
```

The command requires that the user has the 64 bit Cygwin installation and the Arctic Package at the root directory of the C drive. These requirements are not stated anywhere, and left up to the user of the Arctic Package to figure out or to adapt to their needs. In total that gives the original MATLAB implementation a portability score of 1.

8.5.1 Workflow Versions of the Arctic Package

An improvement in installability is that there are now installation instructions for the Arctic Package. Both the Airflow workflow and the KNIME workflow have instructions on how to set up and use the Arctic Package. These are described in Section 6.2.4 for Airflow and Section 6.2.3 for KNIME. The new pipeline also improved installability by compiling and running the Fortran acoustic models in Docker containers. Running them in Docker containers eliminates the need to manually compile them each time the package is installed on a new computer. Instead, there is a Bash script for each model that does it automatically when the models are run for the first time, as described in Section 4.2 and shown in Listing 4.4. The Bash script is run automatically by both the KNIME and Airflow workflows, which means that the user does not have to perform any operations or change anything to compile and run the acoustic models.

Another improvement due to the Docker containers, is that the models can be run without having to adapt the MATLAB code or the Makefiles to the specific system. Instead, the Makefiles are adapted to the Docker containers and their file system and Fortran compiler, which does not change when using different computers. This increases adaptability, as it is easier to install and robust to changes in the environment.

A disadvantage with the workflow versions of the Arctic Package, is that there is more software to be installed and set up, which decreases installability. For the original Arctic Package, the only software needed was MATLAB and a Fortran compiler. For KNIME, MATLAB, Python, Docker, and the KNIME Analytics Platform, must be installed and set up. For Airflow, MATLAB, Python, Node, Airflow, and Docker are required. In addition, the GUI project installs React, RJSF, and Material UI when set up.

One issue with installability that affects the Airflow workflow, is the combination of Airflow and MATLAB. Airflow is not compatible with Windows, but can be used on Windows through Windows Subsystem for Linux (WSL) or with Docker. However, MATLAB is not easy to get up and running on WSL, and running MATLAB in Docker containers requires a network license or the Docker container must enable

a GUI where the user can use *Login Named User* licensing to activate MATLAB. There was not access to a network license and activating MATLAB in Docker was not a practical solution. Consequently, there are currently no instructions to run the workflow on Windows.

Overall, the KNIME workflow has improved portability. The disadvantage to the portability for KNIME, is that more software must be installed to set up the workflow. For the Airflow workflow, it is also a disadvantage that it does not run on Windows. This gives KNIME a portability score of 4, and Airflow a score of 3.

8.6 Maintainability

Maintainability is how easy the system can be modified. The sub-factors in maintainability specified by ISO/IEC 25010 are modularity, reusabiliy, analyzability, modifiability, and testability. In [93], Baggen et al. describes a method developed by the Software Improvement Group (SIG) to measure software maintainability using standardized measurement procedure based on the ISO/IEC 9126 standard [93]. The ISO/IEC 25010 standard replaced the ISO/IEC 9126 standard in 2011. From the 9126 version to the 25010 version, the maintainability, analyzability, and testability criteria were kept, changeability changed name to modifiability, stability is removed, and modularity and reusability are new sub-characteristics.

SIG chose six source code properties as key metrics to asses quality, where metrics 1-4 were first introduced by Heitlager et al. [94]. The metrics they chose were:

1. Volume, as a larger system requires more effort to maintain.
2. Redundancy, as duplicated code must be maintained everywhere it occurs.
3. Unit size, as the units should be kept small in order to increase understanding and keep to a single responsibility.
4. Complexity, as simple systems are easier to understand and test.
5. Unit interface size, as lots of parameters can be a sign of poor encapsulation.
6. Coupling, as tightly coupled components are harder to change.

They also mapped the metrics to the sub-characteristics of maintainability. Which metric affects which criteria is shown in Table 8.5. The unit interface size criterion is not investigated, as it does not make sense to compare MATLAB scripts without an interface to functions. The scripts also largely use the same amount of parameters as the functions, as the source code was refactored with the intent of not changing content. Modularity and reusability were not described, as they were not part of the ISO/IEC 9126 standard, but can be mapped to the same metrics. Modularity is affected by coupling, as tight coupling makes components depend on each other and changes to one also affects the other. Dromey [87] also identifies loose coupling as a modularity property. Modularity can also be affected by unit size, as a modular unit should only have one responsibility, and large units indicates that the unit has

Table 8.5: Overview of which metrics affects which maintainability sub-criteria based on [93].

Metric	Modularity	Reusability	Analyzability	Modifiability	Testability
Volume			✓		
Redundancy			✓	✓	
Unit size	✓	✓	✓		✓
Complexity		✓		✓	✓
Coupling	✓			✓	

multiple responsibilities [86]. Reusability can be mapped to coupling [86], as independent modules are easier to adapt to new contexts. Unit size and unit complexity can also apply to reusability, as large or complex units are less likely to fit into a new context. The metrics are only measured on the MATLAB source code and the workflow specifications, as the acoustic models and libraries are unchanged in the workflow versions of the Arctic Package.

The original Arctic Package is large and complex, as described in Section 3.2, and the software components use a number of different languages. The software was hard to comprehend and make changes to. Some of the maintainability issues were due to the use of global variables stored in the workspace, and that it was challenging to see how the changes would affect the rest of the Arctic Package. The tight coupling with the GUI also made maintainability harder. These factors gives the original MATLAB implementation a maintainability score of 1.

8.6.1 MATLAB Source Code Changes

Table 8.6 shows the statistics for the original and workflow versions of the MATLAB source code. The statistics does not include statistics from the `m_map` library, nor the MATLAB scripts for plotting Bellhop, which were included with the Bellhop acoustic model. In other words, the statistics only includes the files that were created for the Arctic Package. The measurements performed were the number of files, LoC, total number of lines, duplicated code lines, duplicated sections, average unit size, average McCabe Complexity.

The LoC measurement reports the lines of code excluding comments and blank lines, while the total lines measurement reports all lines, including blank and comment lines. The numbers were calculated using CLOC [68]. Duplicated code lines are lines of code that appear two or more places in the source code and that are at least 75 tokens long. The duplicated code lines-measurement counts all lines that are duplicated, excluding the first time they appear. This means that if a section of 10 code lines appear 5 times in the source code, there is 40 lines of duplicated code. The duplicated-sections measurement counts how many different code sequences that have been duplicated across the source code. The results and examples of changes that was done, are explained below.

Table 8.6: Statistics showing how the MATLAB source code changed from the original Arctic Package to the new implementation.

Metric	**Original MATLAB source code**	**Refactored source code**	**Decrease in %**
Files	140	44	68.6
LoC	7 881	1 573	80.0
Total lines	11 919	2 984	74.5
Duplicated code lines	2 220	0	100
Duplicated sections	66	0	100
Average unit size	56.3	35.4	37.1
Average cyclomatic complexity	8.0	7.2	10

Source Code Volume

The source code volume has decreased with 80% from 7881 LoC to 1573 LoC. The decrease is partly due to removing the GUI, functionality that is not added to the workflow versions, and removing duplicated code. 2220 LoC are removed from duplication, and 1518 LoC are removed when decoupling the GUI. Removing the GUI probably caused an additional decrease in LoC, as the 1518 LoC only counts the pure GUI files, and not GUI operations intertwined with processing and plotting of data. Included in the GUI files are scripts used to handle interactions with the map, such as adding, removing, and moving sources. This could be counted as functionality not implemented as well, but as it is possible to add and remove sources through the input files, these are counted as GUI functionality. The identified missing functionality in the new version of the Arctic Package makes up 1244 LoC in the original package. The included and excluded functionality compared to the original Arctic Package is further described by the functional suitability criterion in Section 8.8.

The positive decrease in LoC makes up about 3700 LoC from removing the GUI and duplicated code. Another positive factor that contributes to the decrease in LoC, is that some files were unused, and were not included in the workflow versions of the Arctic Package. Some of the source code was also inaccessible (dead code), due to hard coded values such as `if 1==2`, or `modeflag=0`, and later statements such as `if modeflag == 1`. The negative decrease in LoC, is caused by functionality there was not time to include in the workflow versions.

The LoC values for functionality that is not included in the workflow versions and GUI functionality, are estimates based on grouping files by their purpose. Manual errors when grouping files are possible, as some files can fit into more than one category and files could have been missed. There is also an overlap between the missing functionality and duplicated code, as the functionality that is not implemented has not been refactored to remove duplication.

With regards to source code volume, the new implementation of the Arctic Package is significantly smaller than the original implementation. A lot of the decrease is due to removing code duplication, the original GUI, and unused code. However, a

portion is also due to functionality that is not included in the workflow version of the Arctic Package, because the time frame.

Code Duplication

To measure code duplication over the original and workflow version of the Arctic Package, PMD's Copy-Paste-Detector (CPD) [95] was used. With a threshold of minimum 75 tokens for stating that code was duplicated, CPD found 66 duplicated sections of code in the original version of the package. 15 of the occurrences were duplicated in three or more files, with the most occurrences of a duplicated section of 6 times. The sections varied in length from 7 to 104 lines, with a total of 2220 lines being duplicated across the package, not counting their first occurrence as a duplication. In the workflow version of the package, the analyzer did not find any occurrences of duplicated code larger than 75 tokens. The largest code duplication found in the new MATLAB source code was of 59 tokens. After that there were none until 33 tokens, at which point non duplicated items, such as function calls and if statements started to show up as duplicated.

Listing 8.5 and Listing 8.6 show an example of how code duplication was removed. Listing 8.5 shows the code for loading the bathymetry in the original package, which was part of the script `plotBath.m`. Listing 8.6 shows the code for loading bathymetry in the workflow version of the package where the loading is a separate function `load_bathymetry.m`. In the new bathymetry loading, the code commented out is removed from the function. In addition, the code has been refactored to remove as much of the duplicate code as possible. This shortens the code from 45 lines to 23. For an example, lines 14-27 in Listing 8.5 repeats themselves exactly from line 31-44. The only difference is the names of the loaded variables, which are `lon0`, `lat0`, and `ZZZ0` for the `IBCAO_Arctic_20min.mat` file and `lon`, `lat`, and `ZZZ` for the `IBCAO_Arctic_2min.mat` and `IBCAO_Arctic_30arcsec.mat` files. To account for the variable name differences, the files are loaded into variables with the names `lon`, `lat`, and `ZZZ` for both files (line 3-7 in Listing 8.6), and the new names are used for processing.

Unit Size

Unit size can be measured in lines of code per unit [93]. To find the average unit size, the total line count was divided by the total file count. For the original Arctic Package that gives 7881 LoC total from 140 MATLAB files, which gives a unit size of 56.3 LoC. In the workflow version there are 44 files with a total of 1573 LoC, which gives a unit size of 35.0 LoC. The max LoC in one file in the original version is 340, while in the workflow version it is 145.

One of the reasons unit size has decreased is that large MATLAB scripts have been refactored into functions. For example, the `plotBath.m` and `plotBath2.m` described in Section 4.3, had multiple responsibilities in the original package. The `plotBath.m` script had the responsibility to load the chosen database and process the data for plotting, and it specifies the titles and axis labels to be plotted on the map. In the workflow version, there is one file for each database, where each of them loads and

```
1      qnq=get(h_bathy0,'Value');
2      %   if qnq==2,
3      %     hry='Y';
4      %   else
5      %     hry='N';
6      %   end
7
8     if qnq==2 || qnq==3
9         if qnq==3 % 30sec
10         load ([dataBaseDir slsh 'IBCAO_Arctic_30arcsec.mat'])
11        else
12         load ([dataBaseDir slsh 'IBCAO_Arctic_2min.mat'])
13        end
14         if maxlon > 180
15           lon=[lon lon+360];
16           ZZZ=[ZZZ ZZZ];
17         end
18         if minlon < -180
19           lon=[lon-360 lon];
20           ZZZ=[ZZZ ZZZ];
21         end
22         Ix=(minlon < lon) & (lon < maxlon);
23         Iy=(minlat < lat) & (lat < maxlat);
24
25         earth=ZZZ(Iy,Ix);
26         lonE=lon(Ix);
27         latE=lat(Iy);
28         disp('Please wait...high resolution may be slow...')
29    elseif qnq==1,
30        load ([dataBaseDir slsh 'IBCAO_Arctic_20min.mat'])
31         if maxlon > 180,
32           lon0=[lon0 lon0+360];
33           ZZZ0=[ZZZ0 ZZZ0];
34         end
35         if minlon < -180;
36           lon0=[lon0-360 lon0];
37           ZZZ0=[ZZZ0 ZZZ0];
38         end
39         Ix=(minlon < lon0) & (lon0 < maxlon);
40         Iy=(minlat < lat0) & (lat0 < maxlat);
41
42         earth=ZZZ0(Iy,Ix);
43         lonE=lon0(Ix);
44         latE=lat0(Iy);
45     end
```

Listing 8.5: A section of a MATLAB script from the original package that loads the bathymetry. The `qnq` variable is fetched from the GUI and specifies how to color the map.

```
1     function [lonE, latE, earth, minDpt] =
   ↪  load_bathymetry(bathymetry_file, minlon, maxlon, minlat, maxlat)
2     bath = load(bathymetry_file);
3     if isfield(bath, 'lon') && isfield(bath, 'lat') && isfield(bath,
      ↪  'ZZZ')
4         lon = bath.lon; lat = bath.lat; ZZZ = bath.ZZZ;
5     elseif  isfield(bath, 'lon0') && isfield(bath, 'lat0') &&
      ↪  isfield(bath, 'ZZZ0')
6         lon = bath.lon0; lat = bath.lat0; ZZZ = bath.ZZZ0;
7     end
8
9     if maxlon > 180
10        lon=[lon lon+360];
11        ZZZ=[ZZZ ZZZ];
12    end
13    if minlon < -180
14        lon=[lon-360 lon];
15        ZZZ=[ZZZ ZZZ];
16    end
17    Ix=(minlon < lon) & (lon < maxlon);
18    Iy=(minlat < lat) & (lat < maxlat);
19
20    earth=ZZZ(Iy,Ix);
21    lonE=lon(Ix);
22    latE=lat(Iy);
23    end
```

Listing 8.6: A section of the MATLAB function `load_bathymetry.m` that loads bathymetry in the workflow version of the package.

processes data from that database. Furthermore, a section of reoccurring code that masked values in shallow regions of the map was also extracted to a separate function called `mask_map()`. Before the refactoring `plotBath.m` had a unit size of 198 LoC and `plotBath2.m` had a unit size of 82 LoC. The workflow version has an average unit size of 27.5 LoC for the files `plotBath.m` and `plotBath2.m` were split into. The total LoC is 220, and maximum lines in one file was 80 LoC. In comparison, the total LoC for the files in the original package was 280 LoC and maximum LoC in one file was 198.

There is a clear decrease in average unit size in the MATLAB source code. One of the reasons is that duplicated code is factored out to separate functions or refactored to avoid duplication. Another reason is that large files are split into smaller files with fewer responsibilities. A third factor is that some of the largest files implement functionality related to KRAKEN and timesteps, which is not yet implemented the workflow version of the Arctic Package. However, the refactoring of other large files shows that it is possible to split up the large files, which indicates that it is likely possible to split up the timestep and KRAKEN scripts as well.

Cyclomatic Complexity

Baggen et al. [93] describes a model that uses cyclomatic complexity [96] per unit to measure unit complexity. The cyclomatic complexity is a measure of linearly independent paths within the source code, which means that it measures control flow statements in the source code. The built in Code Analyzer in MATLAB was used to compute the cyclomatic complexity. 1-10 is low complexity, 11-20 is moderate complexity, while 21-50 is high complexity [93].

The complexities of the MATLAB portion of the original Arctic Package and the workflow versions of the Arctic Package are shown by the histograms in Figure 8.6 and Figure 8.7. Figure 8.6 shows the complexity for each file in the Arctic Package, and Figure 8.7 shows the same complexities normalized by the total number of files. The mean complexity in the original package is 8.0, while in the workflow version of the package it is 7.2, which is a small difference. However, the max complexity in the workflow version of the package is 39 and in the original package it was 75. There was also a total of 6 files with a complexity of 39 or higher in the original package. The median complexity is 4.5 for the original package and 4 for the new.

In total, the cyclomatic complexity has changed a little, but both the original and the workflow version of the MATLAB source code falls under the low complexity category for average and median complexity. However, the most complex files in the original MATLAB implementation are removed and the maximum complexity is decreased from 75 to 39.

Coupling

Modularity has improved with the introduction of scientific workflows. The main modularity improvement comes from decoupling the GUI from the MATLAB source code. Using the workflow version of the Arctic Package, the GUI can be replaced without changing anything in the MATLAB source code. Before rewriting the Arctic Package, statements such as `delR = str2num (get (h901, 'String'));` and `freq`

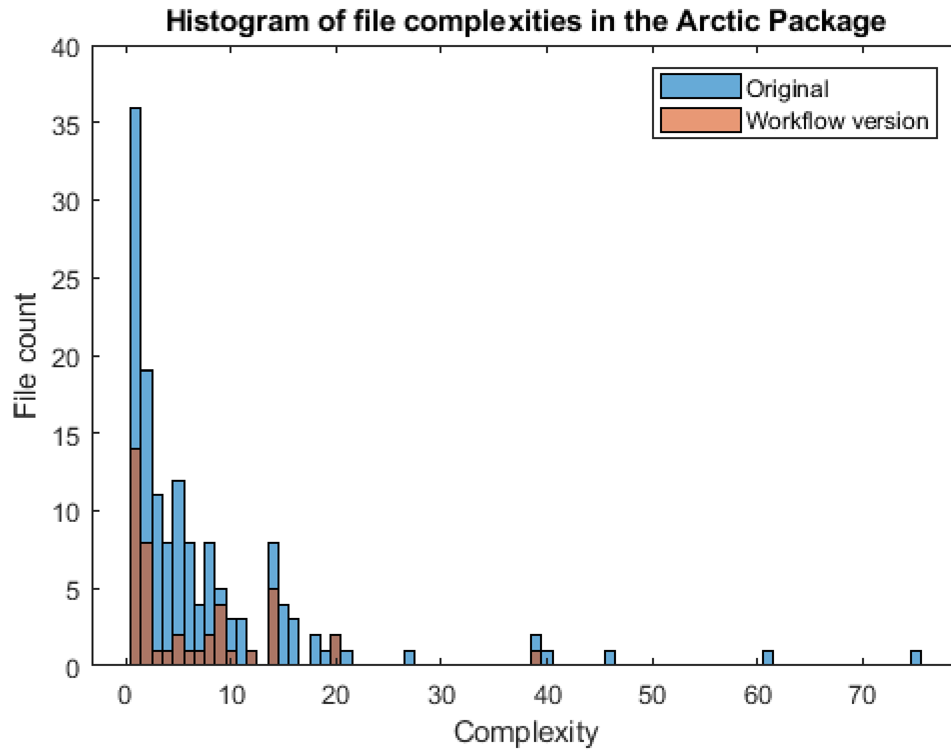


Figure 8.6: Histogram of the complexity of the Arctic Package.

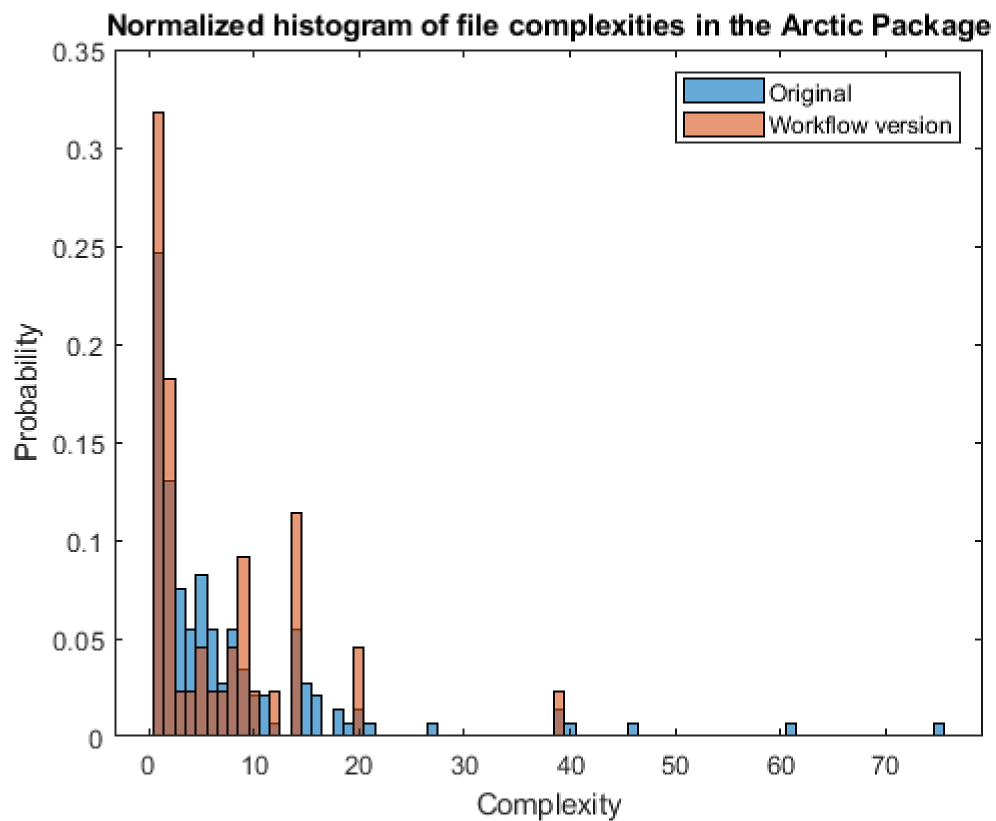


Figure 8.7: Histogram of the complexity of the Arctic Package normalized by total number of files.

`= str2num(get(freqInput, 'String'));` that fetches user input from the main GUI, were located in almost all MATLAB scripts, and connected every file to the GUI. Therefore, to replace the GUI in the original version, all of the statements fetching values from the GUI, would have to be replaced with corresponding values from the new GUI.

Rewriting the MATLAB scripts to functions, also decreases coupling. The `plotBath.m` script was coupled with the `plotBath2.m` script, because it was launched from the `plotBath.m` script. Variables initialized in `plotBath.m` were used in `plotBath2.m`, such as `BAR_LABEL`, `ln` (longitude), `lt` (latitude), and `rd` (radius). The `plotBath2.m` script also had multiple responsibilities that were to plot the coastline, the coloring of the map, sources, and receivers. The new version of the MATLAB code decouples the loading of data in `plotBath.m`, from the plotting operations in `plotBath2.m`. The plotting of sources and receivers is extracted to a separate function that plots them onto a specified map figure. The plotting of the map and coastline is handled by a function `plot_map()`. To load the data, the function `plot_map()` calls a loading function for the chosen dataset.

8.6.2 KNIME workflow

For the KNIME workflow, it is relevant to investigate the metrics volume and coupling. Measuring volume takes into account the code in the MATLAB nodes that needs to be maintained. Measuring coupling investigates whether the workflow GUI solution has loosened the coupling in the Arctic Package. Redundancy is not relevant to measure for the KNIME workflow specifically, as it calls the functions and runs the models, but does not do any processing. The same is true for complexity and unit size. As all the nodes have low complexity due to each node only containing few lines of code to prepare the workspace and run the functions, considering unit size and complexity over the entire workflow would contribute to artificially lower the complexity and unit size for the KNIME workflow.

To assess volume in KNIME, the LoC from the configuration views are measured. There are 233 LoC and 338 total lines over 20 nodes. Considering that the KNIME workflow also generates the GUI that in the original Arctic Package contributed 1518 LoC to the code, that is a large decrease in LoC.

With regards to coupling in the KNIME workflow, the GUI is decoupled from the MATLAB code, but is coupled with the KNIME workflow as the KNIME nodes generate the GUI automatically. There are ways to decouple the GUI from the KNIME workflow as well, for example by writing user input to a file and using a file reader node to read user input to the workflow. That would require some changes to the KNIME workflow, such as removing the input nodes (Section 4.2 describes how the KNIME user input view is created), and making sure the user input has the same variable names as used in the workflow. This approach can be used to take user input using the Airflow GUI, by writing the input to a file that the KNIME workflow reads. The input would likely need some processing, as the JSON output is tailored to the Airflow workflow.

Taking into account both the changes to the MATLAB source code, and the KNIME specific inclusions, the maintainability of the KNIME workflow is assessed to be 5.

8.6.3 Airflow workflow

The same metrics as for the KNIME workflow, volume and coupling, are also relevant to investigate for the Airflow workflow. The GUI and workflow specification both add source code to the package and coupling investigates how the Airflow workflow model and GUI affects coupling.

For the Airflow workflow, the workflow specification contains 246 LoC, and the GUI contains 716 LoC when counting all JavaScript and CSS code. For the Airflow workflow, the total LoC is 962 and total line count is 1019 over 9 files. The volume is heavily influenced by the 613 LoC long parameter specification used to generate the GUI. However, the 962 LoC for the generated GUI is still an improvement from the 1518 LoC used by the original Arctic Package.

With regards to coupling in the Airflow workflow, the GUI is completely decoupled from the workflow. The GUI is a separate application that communicates with the workflow through a REST-API, as described in Section 7.3. The workflow extracts the user input from the JSON payload in the DAG execution requests sent from the GUI applicatoin and uses the input in the workflow.

Taking into account both the changes to the MATLAB source code, and the Airflow specific inclusions, the maintainability of the Airflow workflow is assesed to be 5.

8.7 Compatibility

Compatibility consists of the two sub-factors co-existence and interoperability. Interoperability is how well systems can exchange and use exchanged information and co-existence is how well systems work while sharing resources and environment with other products. In the Arctic Package, interoperability is how the MATLAB processing scripts interacts with the acoustic models, while co-existence is how well other programs work when the workflow is running.

Co-existence is related to performance and reliability. If the Arctic Package consumes a lot of resources or if the system experiences crashes, then co-existence is poor. Section 8.3 describes the reliability of the Arctic Package and Section 8.9 contains details about resources consumed.

8.7.1 Original Arctic Package

The original Arctic Package has some degree of interoperability. The MATLAB scripts creates input files to the Fortran acoustic models, which the model use as input. There are also scripts that read and process the output files from the acoustic models. This means that there is interoperability between the MATLAB scripts and the acoustic models.

Co-existence works well in the original Arctic Package. It consists of MATLAB and Acoustic Models, which consume some resources, but the system does not experience hangs or crashes. Overall the compatibility score for the original Arctic Package is assessed to be 3.

8.7.2 Workflow Implementations

The interoperability is increased in the KNIME and Airflow workflows, due to the refactoring of the MATLAB source code. The MATLAB scripts are rewritten to functions, which are called from the KNIME and Airflow workflows. The interactions between the workflows and the MATLAB functions are described in Chapter 4 and Chapter 5. The interoperability through the input and output files of the acoustic models is kept in the workflow version as well. The Airflow workflow also has increased interoperability through the REST API (Section 5.3), which enables the GUI to communicate with the workflow and trigger DAG executions.

Co-existence has decreased in both Airflow and KNIME, as both system use more resources than the original Arctic Package (Section 8.9). This is especially true when running the Airflow workflow with parallel execution, as the system becomes noticeably slower and has a tendency to hang.

For the Airflow and KNIME workflow models, the compatibility score is assessed to be 3. The interoperability is improved, but the co-existence has decreased, causing the score to be unchanged from the original Arctic Package.

8.8 Functional Suitability

The functional suitability criteria from the ISO/IEC 25010 standard consist of the three sub-characteristics functional correctness, functional completeness, and functional appropriateness. Functional completeness measures how much of the functionality is implemented, functional correctness is whether or not the results are correct, and functional appropriateness is how well the system can perform the tasks. As a goal of the project was to keep the functionality unchanged, it is assumed that the functional appropriateness also remains the same, and thus it is not considered in this evaluation.

8.8.1 Functional Completeness

Functional completeness has decreased in the workflow implementation of the Arctic Package. There was not enough time to rewrite the complete package to fit into the workflow context. The workflows can be expanded by creating functions for the remaining functionality and adding it to the Airflow and KNIME workflows.

15 of the 16 requirements specified in Section 3.4 describe functionality. Of these, 13 are fully implemented in the original Arctic Package and one is partly implemented. For the original package, the functional completeness is $13/15 \approx 87\%$.

The KNIME and Airflow workflows implement an equal amount of requirements. As there was not enough time to implement all of the requirements, the core functionality was prioritized. The workflows implement 9 of the requirements (1-8 and 11), but without the KRAKEN acoustic model, which corresponds to 1/5 of the requirements 5-7. In total that gives a coverage of 8.4/15 = 56%.

8.8.2 Functional Correctness

Functional correctness is hard to measure and compare to the original package. The acoustic models and libraries are unchanged, and hence the functional correctness for those is also unchanged. There were no tests for the MATLAB source code, and it is not possible to say for certain that the refactoring did not introduce any bugs to the software. There were some bugs found and removed as mentioned in previous sections, which increased functional correctness. These were:

- A bug where depths for the Ocean Model dataset in the GUI were different from the values used.
- Ocean Model and GECCO2007 produced NaN values when modeling with them.

Additionally the `Plot paths` button in the GUI sometimes show `Remove paths` when no paths are plotted. This bug is part of the GUI and was removed when it was replaced.

The modeling and processing of results are unchanged except for refactoring, which should be an indication that functional correctness did not change. The various plots were compared to plots produced with the same parameters in the original package, and no mismatches were found. Examples of plots from the original and new implementations are shown in Figure 8.10 and Figure 8.11.

Overall, the original Arctic Package receives a functionality score of 4, as 87% of the functionality is implemented, but there are some bugs. The KNIME and Airflow workflow implements 56% of the functionality and the correctness is slightly increased. This gives both workflows versions a functionality score of 2.

8.9 Performance Efficiency

To measure performance, execution time and memory usage were measured. Time was measured in seconds from the start of a workflow to the end time of the workflow. Memory usage was measured as total memory used in MB when running the workflows. Separate processes could be measured as well, but measuring processes quickly became complex, as it was hard to track exactly which processes that belonged to the workflows at a given time.

The input parameters used when running the tests are shown in Table 8.7. All tests were run on a PC with Ubuntu 18.04 LTS as operating system. The processor was an Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz processor. Total memory is 7913 MB. Figure 8.12 shows the memory usage on the system without running any processes

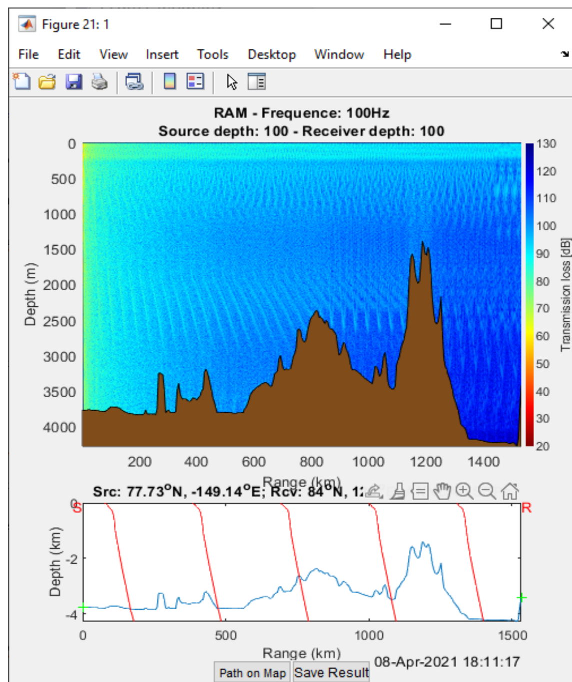


Figure 8.8: RAM original result visualization (top) with temperature profiles and bathymetry (bottom)

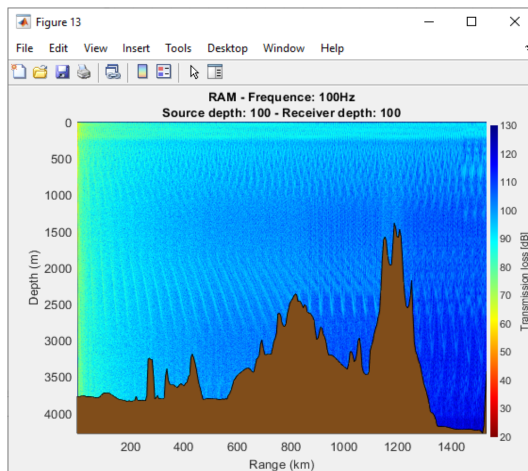


Figure 8.9: RAM result visualization produced by the workflows.

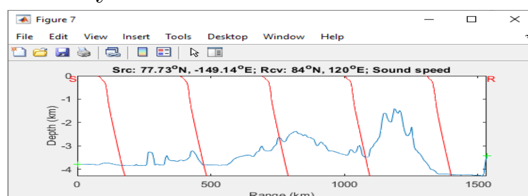
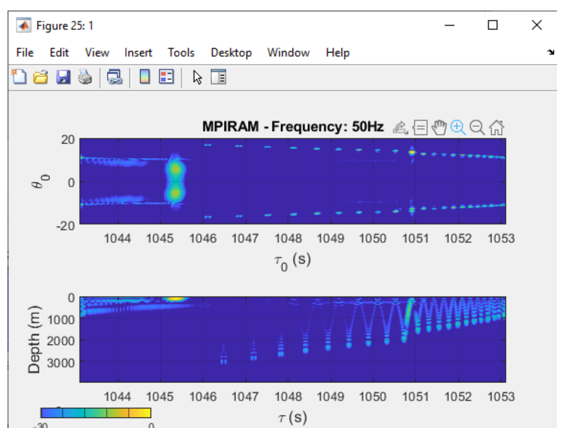
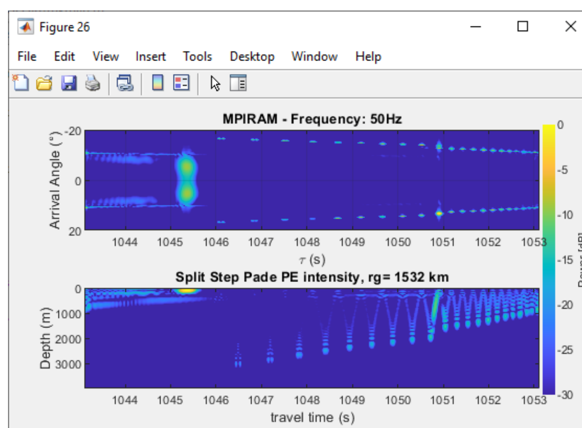


Figure 8.10: Temperature profiles and bathymetry produced by the workflows.



(a)



(b)

Figure 8.11: (a) Shows the visualization of MPIRAM results in the original package, excluding the profile plot shown beneath the figures. (b) shows the visualization of MPIRAM results in the workflow version.

Table 8.7: User input parameters to the workflow for measuring performance

General parameters	
Coloring database	IBCAO - 2 arc minutes resolution
Coastline resolution	Low
Longitude (min, max, center)	-180°E, 180°E, -5°E
Latitude (min, max, center)	70°N, 90°N, 85°N
Radius	15
Plotted sources	10
Plotted receivers	8
Map shape	Rectangular
Selected source (°E, °N, depth)	148.1, 82.2, 100
Selected receiver (°E, °N, depth)	-149.1, 77.7, 100
Sound speed interval (km)	10
Bathymetry interval (km)	5
Sound speed database	ECCOv4
Profile type	Sound speed
Time variation	Annual
Models selected	RAM, MPIRAM, Bellhop, Eigenray
RAM parameters	
Frequency (Hz)	100
MPIRAM parameters	
Frequency (Hz)	50
Q-value	5
Time window width	10
Bellhop parameters	
Model options	Raytracing, incoherent transmission loss
Frequency (Hz)	50
Number of rays	50
Eigenray parameters	
Mode	Eigenrays
Number of rays	2500
Use bottom	Yes
Save paths	Yes
Maximum bottom reflections	3
Angle range	[-10, 10]
Epsilon value	$1 * 10^{-5}$

except for measuring memory and CPU. The command used to measure memory was: `free -m`, which was run every second, and the command to measure CPU was `top -bn2 -d 0,1`, which was also measured every second. The `-b` option indicates that top is run in batch mode, the `-n2` option specifies the maximum number of iterations top should produce before ending, and the `-d 0,1` option sets the screen delay for updates to be 0.1 seconds.

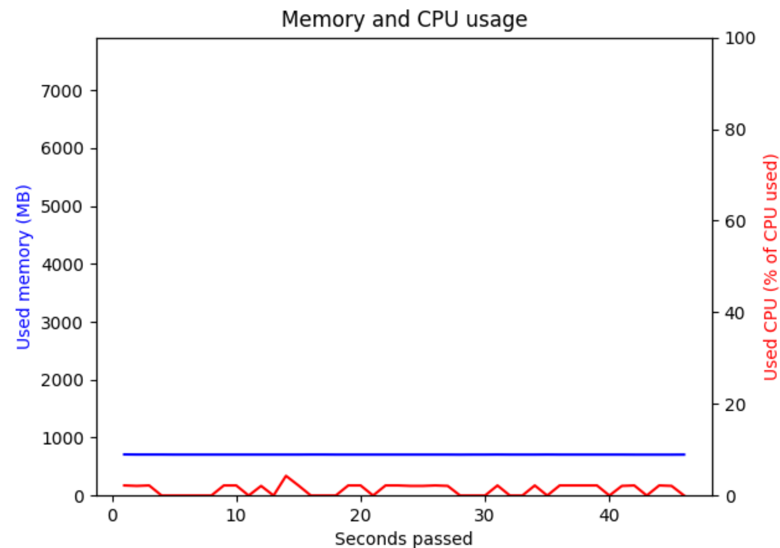


Figure 8.12: Memory and CPU usage on the PC used for testing when no other processes than measuring memory and CPU consumption were running.

8.9.1 Original Arctic Package

The Arctic Package running time from five runs are shown in Table 8.8. They were timed using MATLAB's `Tic` and `Toc` functions. As the program needs user input at multiple different points, the timing was done for each computation part of the code, excluding waiting for input. The *Other tasks* row in Table 8.8 is blank, because the tasks not part of processing functions were hard to measure. These were small functions intertwined with the GUI implementation and also the drawing of the GUI and sub-GUIs. In reality there is some time spent there, but not enough to give a significant change in run time. It is likely at most a few seconds of time spent there in total. Figure 8.13 shows the memory and CPU used when running the original Arctic Package. Resource consumption is measured every second.

The original Arctic Package has a reasonable performance and resource consumption, which gives the original Arctic Package a performance score of 5.

8.9.2 KNIME Workflow

The execution time of the KNIME workflow was measured using Vernalis' benchmarking nodes [81]. The benchmarking nodes records the execution time of each node and the total execution time. Five runs through the workflow were used to

Table 8.8: Execution times in the original workflow, rounded to one decimal

Component	Mean execution time (s)	Max execution time (s)	Minimum execution time (s)	Variance
Map	8.5	8.9	8.3	0.1
Prepare files and plot profiles	7.4	15.5	4.8	20.6
RAM	2.8	2.9	2.8	0.0
MPIRAM	50.4	49.8	52.1	1.0
Bellhop	31.3	32.8	30.4	0.0
Eigenray	8.7	8.9	8.6	1.4
Other tasks	-	-	-	-
Full workflow	**109.2**	**115.59**	**106.5**	**13.4**

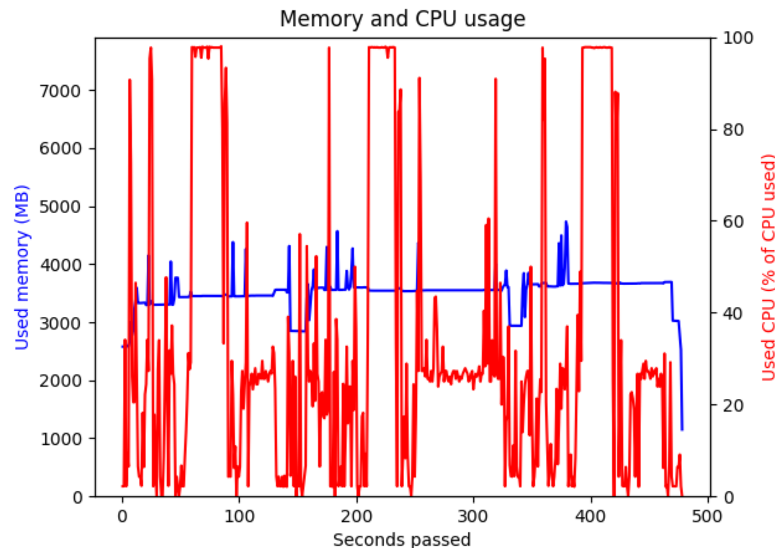


Figure 8.13: Memory usage (blue line) and CPU usage (red line) on the computer used for testing when the original Arctic Package was running.

obtain the running time values and the results are shown in Table 8.9. The large outlier from the `Prepare files` node, is because the first MATLAB node in the first run starts a MATLAB session. The column "Variance run 2-5" shows the variance without considering the first run. Note that the minimum execution times for the nodes does not add up to the total minimum execution time, because the fastest execution time of a node does not always occur in the fastest total run. The same is true for the slowest run.

The KNIME workflow uses on average 2 minutes and 5 seconds to run the workflow. The original implementation used 1 minute and 49 seconds. Some slowdown is expected, as there are new layers of software added both with Docker and with KNIME. However, there are also elements that can contribute to increase performance, for

Table 8.9: Execution times in the KNIME workflow, rounded to one decimal.

Component	Mean execution time (s)	Max execution time (s)	Minimum execution time (s)	Variance	Variance run 2-5
Map	20.1	23.4	19.0	3.4	0.0
Prepare files	3.3	10.4	1.4	15.7	0.0
Plot profiles	0.8	1.2	0.7	0.0	0.0
RAM	8.8	12.7	7.7	4.7	0.0
MPIRAM	54.7	55.9	53.8	0.8	0.4
Bellhop	26.2	26.7	25.8	0.1	0.0
Eigenray	11.4	11.2	11.7	0.0	0.0
Other nodes	0.1	0.2	0.1	0.0	0.0
Full workflow	**125.3**	**142.1**	**125.4**	**87.9**	**0.2**

example are the tasks ”prepare files” and ”plot profiles” run before each of the acoustic models in the original Arctic Package, but only once per workflow run.

The first time the performance was measured, the workflow used around 60 seconds on average to run the map task, and the Eigenray model used almost 2 minutes. These slowdowns contributed to the workflow using about 6 minutes to complete. The slowdown was investigated, and for the map the reason was that saving and loading figures takes a lot of time. In the map component a figure was saved for each of the four plotting operations (plot coloring, plot sources, plot receivers, plot paths), which each adds about 10 seconds to the execution time. To reduce the slowdown, the plotting of the map was changed to be performed in a single node, thus eliminating the need to save and load figures. The change decreased the run time of the map from 60 seconds to 20 seconds on average.

The slowdown in the Eigenray model only occured when running the model in the Eigenray container. To speed it up, the base image in the Docker container was replaced, and the flags for the Fortran compilation were changed to match the Makefile used in the original Arctic Package. The updated Docker container improved the Eigenray execution time in the workflow from 2 minutes to 20 seconds.

In total, the map and Eigenray performance improvements reduced the execution time by 2 minutes. In theory, this means that the workflow should use four minutes to run. Instead, it only uses two minutes. A possible explanation is that the workflow crashed a couple of times the first time the workflow was benchmarked. The benchmarking was restarted after a crash, but the slow execution time and crashing are likely to be related as they occurred together. However, as the workflow was the only application running on the computer, it is hard to tell exactly what caused the issue. A possible explanation is that the improved execution time for the other tasks might be an unexpected benefit of the optimizations for the map component and

Eigenray.

In theory, KNIME executes independent nodes in parallel. However, KNIME launches a MATLAB instance when executing MATLAB nodes, and all of the MATLAB nodes use the same instance to run. In practice, this causes them to run sequentially. This means that only the independent nodes that are not MATLAB Snippet Nodes can be run in parallel, leading to the workflow being almost entirely sequential.

The memory and CPU used by the KNIME workflow is shown in Figure 8.14. The workflow is executed three times. In the memory graph, the three runs are visible by the two dips in memory usage. It is also visible by the larger spikes in CPU usage. The processes that consume the most memory is Java (as KNIME is written in Java) and MATLAB.

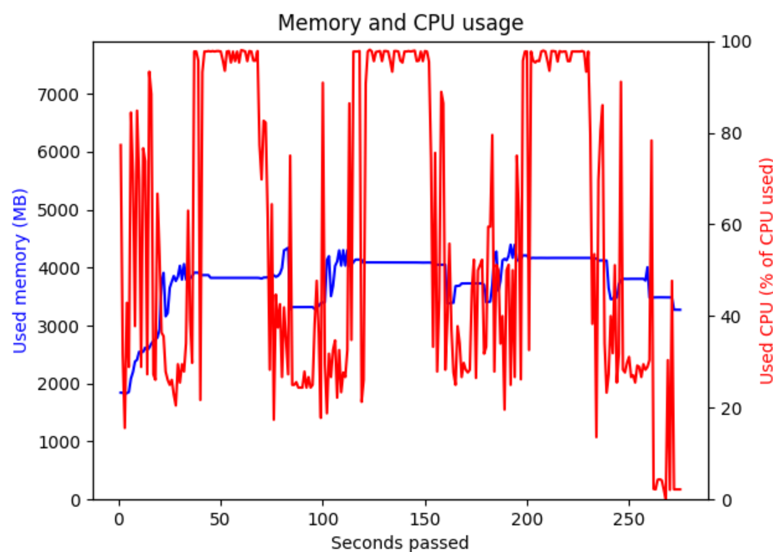


Figure 8.14: Memory (blue line) and CPU usage (red line) on the computer used for testing when the KNIME workflow was running. The workflow was run three times, and then MATLAB was closed.

The KNIME workflow is slightly slower than the original Arctic Package, about 15 seconds on average. It uses some more CPU than the original, but not enough to be an issue. In total the KNIME workflow receives a performance score of 5, as it for all practical purposes perform as well as the original package.

8.9.3 Airflow Workflow

The execution time of the Airflow workflow is measured by Airflow. The workflow was measured when running sequentially using the SQLite database and in parallel using the PostgreSQL database. The results are shown in Table 8.10, and are Airflow's reported times for each of the tasks. The task times reported are from the sequential run of the workflow. The last row shows run times for a parallel runs using PostgreSQL, which allows multiple tasks to be run at the same time.

Table 8.10: Execution times in the Airflow workflow, rounded to one decimal.

Component	Mean execution time (s)	Max execution time (s)	Minimum execution time (s)	Variance
Map	22.8	23.9	22.3	0.4
Prepare files	7.7	7.9	7.5	0.0
Plot profiles	12.5	13.4	12.2	0.2
RAM	23.1	23.9	22.3	0.4
MPIRAM	73.5	75.0	71.5	2.2
Bellhop	54.8	56.3	53.8	1.0
Eigenray	30.3	32.0	29.0	1.3
Other tasks	0.5	0.6	0.5	0.0
Sequential total	**249.0**	**252.0**	**245.7**	**5.8**
Parallel total	**159.8**	**170**	**149**	**74.7**

The Airflow workflow also benefited from the changes made to speed up the plotting of the map and the Eigenray model. The map used around 60 seconds in Airflow before the change and now uses 20 seconds. The Eigenray model used around 1 minute 20 seconds. Now, it uses 35 seconds. The workflow uses on average 4 minutes and 9 seconds to run through the workflow when running it sequentially. Switching the SQLite database with the PostgreSQL database enabled Airflow to run tasks in parallel, which sped up the execution to 2 minutes and 39 seconds on average.

The Airflow workflow uses longer to perform the plotting operations than KNIME. Part of the reason is that Airflow saves the figures to the file system, while KNIME outputs them to a viewer node and avoids the figure saving. This is evident from the `Plot profiles` operation, which takes only a second in KNIME, but 12.5 in Airflow. Airflow also spends time starting MATLAB and exiting MATLAB with each task, while KNIME only spends time staring a MATLAB session the first time a MATLAB Snippet node is run. The startup time to run a MATLAB function in batch mode was measured to 3.9 seconds. The time was measured using the command: `matlab -noFigureWindows -batch -timing "test"`, where `"test"` is a dummy script. The complete log from the startup timing is shown in Appendix A. There are 20 tasks launching MATLAB, which with a startup time of 3.9 seconds uses 78 seconds. The startup overhead can be improved by using fewer tasks, as the startup then happens fewer times, but this has the downside of decreasing the separation and natural structure of the workflow.

On average, when running the workflow sequentially, both the KNIME workflow and the original Arctic Package are over 2 minutes faster than the Airflow workflow. However, the Airflow workflow has the option to parallelize the workflow, which causes the average run time to decrease by 1 minute and 30 seconds. Running the Airflow workflow in parallel is still slower than the KNIME workflow, but only by 34 seconds.

The memory and CPU used by computer when the Airflow workflow was running, is shown by Figure 8.15. Three workflow runs were performed, which can be recognized from the three spiked sections in the graph. The tasks were run in parallel when the measurements were taken. Most of the memory usage was caused by MATLAB. Airflow uses some more resources than KNIME and the original Arctic Package.

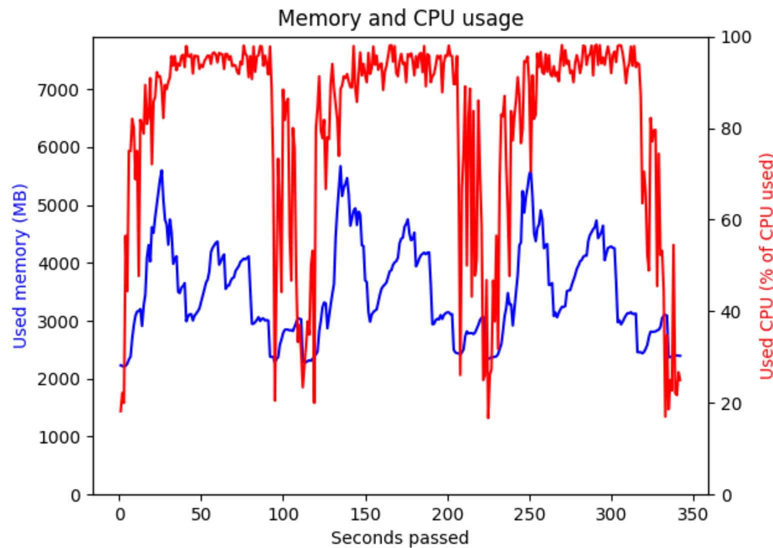


Figure 8.15: Memory usage (blue) and CPU usage (red) on the PC used for testing when the Airflow workflow was running. The workflow was run three times, and then the GUI, Airflow scheduler, and Airflow webserver was closed. The measurements were taken when the workflow was run with parallel execution enabled.

The Airflow workflow is slower than the original Arctic Package and the KNIME workflow, even with parallel execution. However, the performance is satisfactory, which gives Airflow a performance score of 4.

8.9.4 Resource Consumption Comparison

The graphs in Figure 8.16 and Figure 8.17 show, respectively, the memory and CPU used for each of the Arctic Package versions. Note that the original MATLAB implementation takes longer to complete the three runs through the workflow than both the Airflow and KNIME workflows. The reason for that, is that the user has to input values to the workflow at multiple different points in time and that the user spends time selecting sources and receivers on the map for each acoustic model to run. Furthermore, the user input happens in between computations, as interacting with the GUI while a model is running, causes the results to be plotted on the figure that was last interacted with (usually the GUI). For the Airflow and KNIME workflows, all the input is given before starting the workflow, and giving user input does not interfere with the execution of the workflow. If the workflow did not have the same input parameters with each run, the KNIME workflow would also spend some time on user input, as it is not possible to give user input while the workflow is running. Giv-

ing user input to a running workflow in KNIME would result in resetting the workflow.

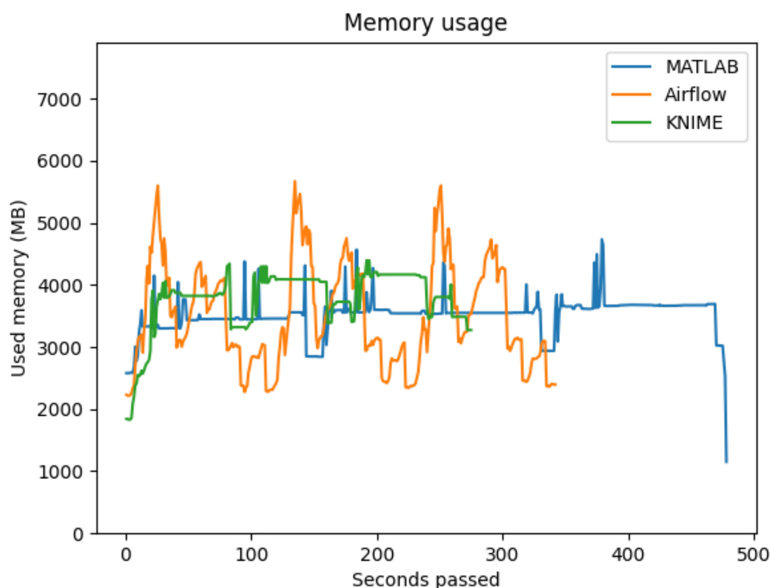


Figure 8.16: The memory usage of the three different versions of the Arctic Package. The blue line is the original MATLAB implementation, the green line is the KNIME workflow, and the orange line is the Airflow workflow. The execution times varies because these values also include the time spent to input values to the Arctic Package.

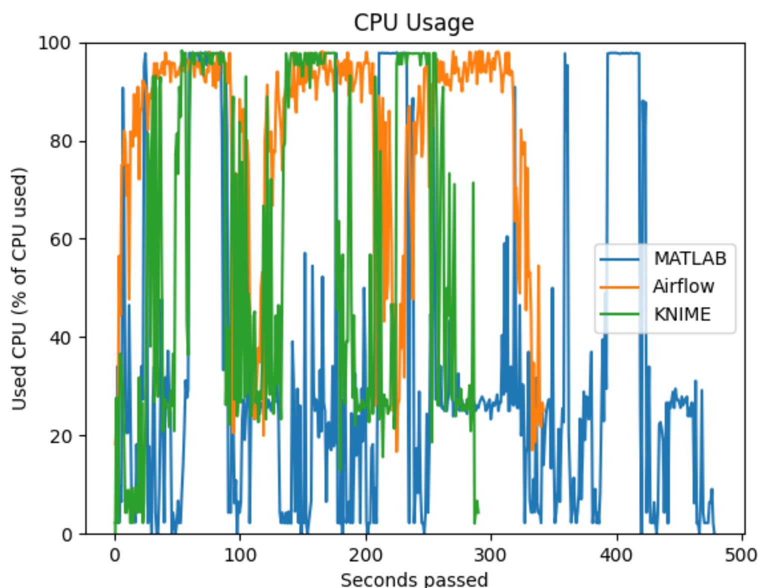


Figure 8.17: The CPU usage of the three different versions of the Arctic Package. The blue line is the original MATLAB implementation, the green line is the KNIME implementation, and the orange line is the Airflow implementation. This was measured at the same time as the memory usage in Figure 8.16.

8.10 Evaluation Summary

The evaluation above discussed changes in the Arctic Package and their impact on the quality criteria. A summary of the scores for each of the Arctic Package versions and the average quality are shown in Table 8.11.

Table 8.11: Evaluation summary and scoring. A score of 1 represents that the implementation poorly satisfies the criterion, while a score of 5 is a satisfactory criterion fulfillment.

Criteria	**Original Arctic Package**	**KNIME Workflow**	**Airflow Workflow**
Reliability	3	1	4
Usability	1	2	4
Portability	1	4	3
Maintainability	1	5	5
Compatibility	3	3	3
Functionality	4	2	2
Performance	5	5	4
Total of 35	**18**	**22**	**25**
Average	**2.6**	**3.1**	**3.6**

The Airflow workflow scores best out of the two workflows, with an average quality score of 3.6 of 5. KNIME has a score of 3.1 of 5, and the original Arctic Package has a score of 2.6 of 5. Both workflows has a lower score for functionality than the original Arctic Package. The KNIME workflow also has a lower score for reliability. While the Airflow workflow has a lower score for performance. The main increase in software quality common for both workflows, is a consequence of the increase in maintainability, usability, and portability. The Airflow workflow also has an increase in reliability.

The low score for functionality is to be expected, due to the time frame of this master project. To refactor the Arctic Package, improve on the project pipeline, create two different workflow models, and generate user interfaces for the workflows, is too much work to finish with the complete set of requirements. However, the functionality score can be improved by refactoring the features not yet added to the workflows and adding them to the workflow models.

The performance will likely be lower for the workflows than the original implementation, considering that there is an extra layer of software in addition to the MATLAB source code. There is, however, possibilities for improvement in the performance, as shown by the optimizations to the map and the Eigenray model.

Another way to improve the performance, especially for the Airflow workflow, is to rewrite the MATLAB source code to another language, for example to Python. The workflow specification in Airflow is written in Python, and there are dedicated Python operators. In the current workflow, a Bash operator was used to call the MATLAB functions, as there were no MATLAB operators. The Bash operator

launches MATLAB in batch mode, but there is some overhead with this approach, as MATLAB needs to start up each time a task is run. The startup time to run a MATLAB function in batch mode was measured to 3.9 seconds (Appendix A), which adds 78 seconds to the run time over the 20 tasks. Consequently, the potential execution time improvement is 78 seconds.

Rewriting the MATLAB source code to Python can also have a positive impact on portability for Airflow, as the entire workflow could be run in Docker without any challenges with MATLAB activation. For the KNIME workflow, rewriting the MATLAB source code to Python, could improve reliability, as the reliability issues seems to be caused by the integration between KNIME and MATLAB. Furthermore, the Python nodes includes more features, such as a customized amount of input and output ports and access to flow variables directly through the scripting configuration window.

Although the security criterion was not discussed previously in the evaluation, there is a potential security issue with the Airflow workflow that should be noted. The input from the GUI is not currently sanitized and inputted directly to the Bash operator. Consequently, the user can input anything directly to Bash in the workflow, and a user with malicious intent can exploit this. The issue would be solved by passing the variables to the Bash operators environment argument, but the attempts to use the operators environment option with MATLAB were unsuccessful due to issues with escaping nested single- and double quoted items. This is not an issue as long as the workflow is only run locally, but if the Airflow workflow is made available to a larger user base, this vulnerability should be addressed.

An element that is not covered by the ISO/IEC 25010 standard, is the difference between the graphical and textual based SWFMSs. While the KNIME workflow specification is based on drag-and-drop programming, the Airflow specification is written in Python. The graphical canvas with drag-and-drop programming has the potential to be simpler to create and expand for scientists inexperienced with programming. However, as all of the MATLAB nodes has to be configured using MATLAB code, a lot of the benefit disappears. Moreover, there is not much room for changes as long as only the workflow specification is changed. The underlying MATLAB functions would have to be changed as well. These factors contribute to minimize the benefit achieved by a graphical workflow specification, as there is a continued need for programming knowledge when developing the Arctic Package, even when using graphical workflow specification.

Another potential benefit from scientific workflows not captured by the evaluation using the ISO/IEC 25010 standard, is the transparency and documentation benefit from the scientific workflow models. The workflow models contribute to show the steps in the processing and visualizations in the Arctic Package, and the DAGs (Figure 4.1 and Figure 5.1) can provide documentation for the code and its structure. Consequently, it might not be necessary to create or generate separate models of the code, for example using UML. Furthermore, the nodes represent tasks performed by the underlying code, which can simplify locating the correct functions, for example to change, debug, or maintain a specific task.

Chapter 9

Conclusions and Future Work

In this thesis, the Arctic Package, a software package for acoustic modeling and visualization, was adapted to a scientific workflow context in an attempt to simplify setup, use, and maintenance of the package. The package has the potential to be a tool that makes acoustic modeling more easily available and to contribute in planning and analysis of acoustic thermometry experiments. However, it was not much used due to a challenging setup, a complicated GUI, and no instructions on how to setup and operate it.

This chapter summarizes the findings presented in previous chapters, and discusses them in the context of the research questions and Design Science Research (DSR). Section 9.1 summarizes the chapters and relates the findings to the research questions and the associated activities, Section 9.2 discusses factors that may have impacted the conclusions and their generalizability, and Section 9.3 discusses future work for the Arctic Package.

9.1 Research Questions Revisited

The five research questions are revisited below. They formed the basis for developing the workflow versions of the Arctic Package, together with the DSR methodology that tied the project to the environment and the existing knowledge base. From the DSR methodology, the five main activities were utilized [28]:

- A1 Explicate the problem, to investigate and analyze a problem and its causes.
- A2 Define requirements, find a solution to the problem and how to develop the artifact to solve it.
- A3 Design and develop the artifact, create the artifact based on the requirements.
- A4 Demonstrate artifact, use it in a real-world situation to demonstrate that it solves or improves on the problem.
- A5 Evaluate artifact, determine how well the solution works and fulfills the requirements.

RQ1 What kinds of scientific workflow management systems exists, and which ones are suited for the Arctic Package?

There exists two main groups of SWFMSs, SWFMSs that use graphical workflow specifications and SWFMSs that use textual workflow specifications. Another possible way to divide workflows, is into groups of control-driven and data-driven workflows. Control-driven workflows focus on the order tasks are executed in and data-driven workflows focus on the data passed between the tasks. To find the SWFMS that was best suited for the Arctic Package, the knowledge from related work (Section 2.4) and from conversations with NERSC about the challenges and requirements for the Arctic Package (Chapter 3), were used. We found that KNIME and Airflow were suited for the Arctic Package, as they are both open source, multi-purpose, regularly updated, and with large user communities. Choosing KNIME and Airflow also includes one representative from each group of SWFMSs. KNIME is data-driven oriented and uses graphical workflow specification, and Airflow is control-driven oriented and uses textual workflow specification.

The answer to RQ1 was obtained through A1, which investigates and analyzes the problems in the Arctic Package. RQ1 uses the rigor cycle in DSR to find information about related projects and gain insight into promising results from SWFMSs used in similar projects. RQ1 also uses the relevance cycle to ensure that the artifact developed addresses the challenges the stakeholders face when using the Arctic Package.

RQ2 How can the computations performed by the Arctic Package be represented by a scientific workflow model? What kind of data and which tasks are required in a scientific workflow model of the Arctic Package?

The computations and data in the Arctic Package were summarized in Figure 3.2. The data used in the package was data describing the coastline, bathymetry, sound speed, temperature, salinity, sources, and receivers. The top-level tasks required in a scientific workflow model of the Arctic Package are to plot a map of the modeling area, to plot the profiles and bathymetry, to prepare the modeling data, and to prepare, run, and visualize each of the acoustic models and their results. In scientific workflows, the tasks are generally represented as DAGs. However, to achieve this structure for the Arctic Package, the source code needed to be refactored. Refactoring included decoupling the GUI from the MATLAB scripts, decoupling the MATLAB scripts from each other, and creating a new GUI separate from the MATLAB source code.

Using the refactored code, the KNIME workflow (Chapter 4) and the Airflow workflow (Chapter 5) were developed. The KNIME workflow was developed in parallel with the refactoring of the MATLAB source code, and then the Airflow workflow was developed using the same refactored MATLAB source code as KNIME. The resulting workflow models are shown in Figure 9.1 and Figure 9.2.

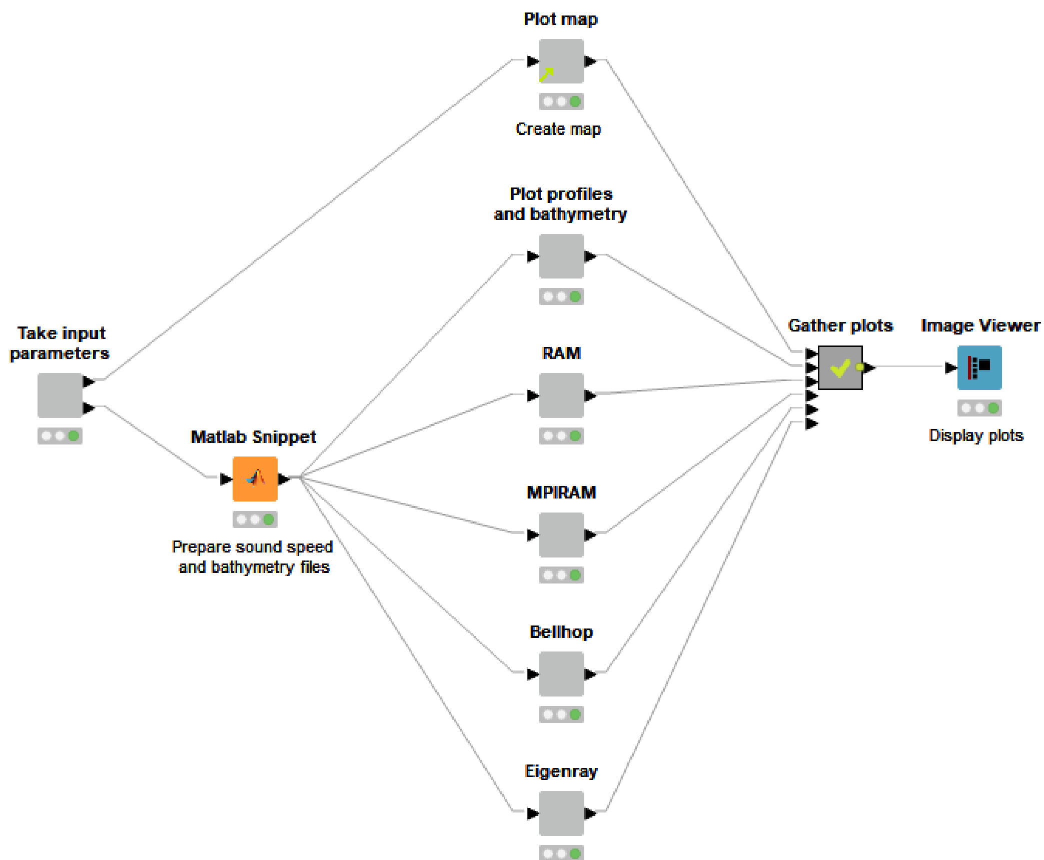


Figure 9.1: The top level KNIME workflow model for the Arctic Package. This figure is a copy of Figure 4.1.

To answer RQ2, A2, A3, and A4 were used. A2 is concerned with developing the requirements for the artifact in order to solve the problem, which answers what kind of data and which tasks that are required in a scientific workflow of the Arctic Package. This process is part of the relevance cycle, as the 16 requirements for the Arctic Package were developed in collaboration with the NERSC stakeholders. A3 is concerned with design and development of the artifact, which answers how the computations can be represented by a scientific workflow. This is part of the rigor cycle as it is based on knowledge from related work that used scientific workflows. The demonstration of the artifact (A4) was performed in parallel with developing the artifact, as demonstration for the NERSC stakeholders provided feedback to improve the artifacts. This process constitutes the design cycle of DSR, but also contributes to the relevance cycle, as the artifact was adapted based on the stakeholders' feedback. A weakness with A4 in this project, is that the feedback was based on digital demonstrations of the application due to the Covid-19 lockdowns.

RQ3 Can a GUI for the Arctic Package be automatically generated based on a specification of the input parameters?

It is possible to automatically generate the GUIs for the Airflow and KNIME workflows. For KNIME it was generated using built-in widget nodes, while the Airflow

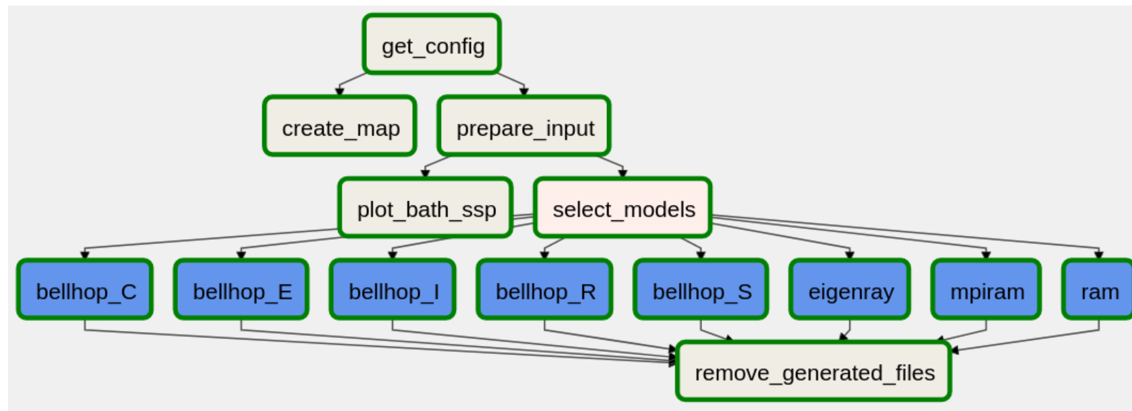


Figure 9.2: The constructed Airflow workflow model for the Arctic Package. This figure is a copy of Figure 5.1.

GUI was developed as a separate React application that generated the GUI using the RJSF librabry. The RJSF library generated the GUI based on a JSON schema that described each of the parameters.

RQ3 was based on A2, and attempts to improve the complicated GUI in the Arctic Package. RQ3 uses the relevance cycle through conversations with the researchers at NERSC to develop an updated GUI for increased usability. The workflow GUIs also contribute to ease maintainability, as the tight coupling is loosened from automatically generated GUIs that do not depend on the MATLAB scripts.

RQ4 How does a DevOps pipeline combined with container technology affect software quality in the context of the Arctic Package?

RQ5 How does scientific workflows affect software quality in the Arctic Package?

RQ4 and RQ5 are related to the evaluation of the Arctic Package. The evaluation measures software quality and was performed using the ISO/IEC 25010 standard. Each of the Arctic Package versions were given a score from one to five for the quality criteria, depending on the degree of satisfaction for the criteria. The criterion satisfaction was estimated based on the impact concrete changes to the Arctic Package had and on results from metrics used for estimating quality for the ISO/IEC 25010 standard and its predecessor the ISO/IEC 9126 standard. The workflows were evaluated using the criteria reliability, usability, portability, maintainability, compatibility, functional suitability, and performance efficiency. The software quality scores for the original Arctic Package and the two workflow versions are shown in Table 9.1.

The DevOps pipeline mainly affected the portability and compatibility quality criteria. The positive impact on portability is evident from the portability score that increased from 1 in the original Arctic Package to 4 in the KNIME workflow and 3 in the Airflow workflow. On the other hand, there is no change for the compatibility criterion. This is because the positive effect from the pipeline on the sub-criterion

Table 9.1: Colored version of the evaluation summary and scoring table (Table 8.11). A score of 1 represents that the implementation poorly satisfies the criterion, while a score of 5 represents that the implementations satisfy the criterion.

Criteria	**Original Arctic Package**	**KNIME Workflow**	**Airflow Workflow**
Reliability	3	1	4
Usability	1	2	4
Portability	1	4	3
Maintainability	1	5	5
Compatibility	3	3	3
Functionality	4	2	2
Performance	5	5	4
Total of 35	**18**	**22**	**25**
Average	**2.6**	**3.1**	**3.6**

interoperability was offset by the negative change in the sub-criterion co-existence, causing the total compatibility score to remain the same.

The scientific workflow versions affected all of the software quality criteria evaluated. Overall, the Airflow workflow received the best average quality score of 3.6 of 5, compared to 3.1 for the KNIME workflow and 2.6 for the original Arctic Package. The Airflow workflow generally scored better than the two alternatives, with only the functionality criterion scoring below 3 with a score of 2. Compared to the original Arctic Package, the Airflow workflow scores better for 4 out of 7 criteria, equal for 1 out of 7, and worse for 2 out of 7. The improved criteria are reliability (+1), usability (+3), portability (+2), and maintainability (+4). Compatibility is equal for both implementations, while the criteria that Airflow scores worse for are functionality (-2) and performance (-1). The difference in performance is small, 50.6 seconds for execution time and some more memory and CPU were used.

The KNIME workflow scores better than the original Arctic Package for 3 out of 7 criteria, unchanged for 2 out of 7 criteria, and worse for 2 out of 7 criteria. Usability (+1), portability (+3), and maintainability (+3) are improved, compatibility and performance is unchanged, and reliability (-2) and functionality (-2) has decreased. Even though the usability score has improved by 1, the score is still low at 2, which indicates that using the package is challenging when using the KNIME workflow implementation.

The answers to RQ4 and RQ5 were based on A5. The evaluation contributes to the relevance cycle by taking into account the stakeholder challenges when evaluating the Arctic Package. In turn, the results from the evaluation contributes to the rigor cycle by providing information to the knowledge base about how a complex software package in the acoustic modeling domain can benefit from scientific workflows and which challenges there are.

The results show that it is easier to use and maintain the Arctic Package with the workflow versions. The package is more modular, has less duplicated code, less

lines of code, and a smaller unit size, all factors that improve maintainability. The workflow versions of the package are also easier to set up, with the exception of Airflow on Windows computers, which is shown by the increased portability score. Another improvement is that the Airflow workflow is easier to use and slightly more reliable. The user encounters few errors, and has an easy and consistent user interface to interact with. On the other hand, the KNIME workflow is not much easier to use and less reliable. The GUI suffers from sub-optimal solutions to work around the static input fields that ideally should have changed options based on selections made by the user.

To summarize, this master project contributes to simplify the process of acoustic modeling in the previously large and complex software package the Arctic Package. The package was adapted to a scientific workflow context and updated with documentation and instructions on how to set up and run the models. There is still remaining work (Section 9.3) to get to an optimal workflow implementation with the complete set of functionality, but the core functionality is implemented and shows promise for increasing the software quality. Consequently, distributing, using, maintaining, and updating the software is likely to be easier in the future. This can contribute to a more widespread use of the package and increase the practical value, for example by contributing to plan and analyze acoustic thermometry experiments.

9.2 Threats to Validity

The user testing of the workflow implementation of the Arctic Package was very limited, due to Covid-19. This causes the results to be based on the quality criteria evaluation of the package alone. Ideally, the package should have been evaluated by its intended users in order to obtain conclusive results on whether or not the package has improved in a scientific workflow format. The steps taken to mitigate this issue, were to have a dialogue with the stakeholders and to demonstrate the application during development. However, these meetings were online most of the time, which limited the possibilities for interaction with the workflows. The limited contact with NERSC also constitutes a weakness with regards to the DSR methodology.

The evaluation of performance was measured using a specific set of parameters. This can skew the results in the case that one of the implementations perform better with parameters not tested with the application. This scenario is not very likely, as the exact same underlying MATLAB functions were used. Moreover, the most time consuming operations were to run the acoustic models, which are run in Docker containers and do not depend on other factors. During the work with the workflows, other parameters were tested without measuring exact performance and no difference in performance was detected. The tests were also on a specific computer running Ubuntu 18.04 LTS with a specific hardware. Another computer or operating system can get different measurements.

There was some variance in the results obtained when running the KNIME workflow. The benchmarking values varied by about 2 minutes when tested on the same computer with the same settings and workflow. The same variance was not observed for the Airflow workflow. As there is some unknown factor affecting the performance

of KNIME, the results may deviate from the actual performance of the workflow. However, the final benchmarking values seem sensible, as the execution times for the model runs are close to the execution times of the models outside the KNIME workflow.

The starting point of this project was a highly coupled and highly complex software package. A smaller application with less source code, looser coupling, and lower complexity might not see the same improvements from applying scientific workflows as this application. Moreover, only a subset of the functionality in the Arctic Package was implemented in the workflow versions, and developing the remaining functionality for the Arctic Package can encounter challenges not considered in the evaluation of this project.

The results obtained from this project are based on a software package consisting of multiple MATLAB scripts, and although possible to adapt for, the SWFMSs were not created for execution of MATLAB source code. This means that a similar projects using another programming language can produce different results, depending on the language support in the SWFMS. Other SWFMSs can also provide different results than KNIME and Airflow. The SWFMSs were selected from the two different categories of SWFMSs, textual and graphical, to investigate what may be the most suitable. However, as there were only one representative from each group, two other SWFMSs can produce different results. Nevertheless, this thesis can give an indication about how well the two different workflow categories fit to an acoustic modeling and visualization application and what the consequences in terms of software quality are.

9.3 Future Work

On short term, the future work for the Arctic Package is to further develop it to implement the remaining of the functionality from the original Arctic Package. The functionality not included in the workflow version is the KRAKEN model, running acoustic models for different timesteps, plotting the results of multiple acoustic models in one plot, visualizing saved results, running acoustic models for multiple receivers, overriding source and receiver depths in the GUI, and running acoustic models for multiple sound speed databases. Considering limitations with regards to reliability and usability for KNIME, Airflow is the best choice for further development.

For the short term future, it can also be useful to perform user testing of the package and test it in a real world environment. This can help investigate whether the software has any weaknesses not picked up by the evaluation using the ISO/IEC 25010 standard. The package can also benefit from creating tests for it and from automating the setup.

Another expansion for the future would be to fetch the datafiles directly from a database, instead of copying them to each computer using the Arctic Package. The files were too large and too many to be included in the GitHub repository, which means they must be sent from user to user or uploaded to a shared folder for new users to access them. In addition to fetching datafiles from a database, it is possible to host the application in the cloud and expand the application pipeline to include

continuous delivery and deployment.

For long term future work, it can be beneficial to rewrite the Arctic Package MATLAB scripts to a non-proprietary language, as MATLAB is proprietary and expensive. The cost makes it is harder to distribute, as every user of the Arctic Package must have a MATLAB license. To adapt the workflow to a new language, the MATLAB function calls from the workflow must be replaced with calls to the functions in the new implementation. The rest of the workflow specification, and the structure and order of the tasks can remain unchanged. If the package is further developed in Airflow, Python can be a natural choice for redeveloping the package, as the workflow specification is in Python. Having both the processing scripts and the workflow specification in the same language reduces the software technology stack, which means that there are fewer languages to learn in order to maintain the package.

List of Listings

2.1 Python code for composing operators to decide their relationship. . . 30
2.2 How to create a BashOperator using Airflow syntax. 31
3.1 How the Bellhop model is started from MATLAB. Lines 1-8 runs the model if the OS is macOS or Linux and lines 9-14 runs the model on Windows. 38
4.1 The MATLAB code in the Plot map node. 53
4.2 The MATLAB code for the Prepare RAM input node from Figure 4.6. 54
4.3 The Run RAM node in Figure 4.6 configuration window. 55
4.4 The Bash script launch_ram.sh that builds the Docker image for the RAM acoustic model if there is no RAM image, and runs the RAM model. 55
4.5 The configuration code of the Collect model result node from Figure 4.6. 56
4.6 The Plot RAM node in Figure 4.6 configuration window. 57
5.1 The Bash operator code for preparing the RAM input files. 63
5.2 The Bash operator code for running the RAM model. The launch_ram script is shown in Listing 4.4. Each model has its own launch script available in [74]. 63
5.3 The Airflow implementation of the branching, where the BranchPythonOperator selects which model branches to run. 64
5.4 The code to create a branch for each model type in the Bellhop acoustic model. 65
5.5 An example of a request body to trigger a DAG. The conf object is generated by RSJF, and the Airflow workflow requires that the input is named and structured as in this example, otherwise the Airflow workflow will not recognize the variables. The (...) section indicate that a part of the object was left out for brevity. 66
5.6 Docker commands to start the Docker container that runs the Bellhop model. 67
6.1 The contents of the Dockerfile for the RAM acoustic model. 71
6.2 The configuration code for the Plot RAM node in Figure 4.6. 73
6.3 The settings file containing path information about the Arctic Package folders. 77
6.4 Command for creating an Airflow user. Replace the angle brackets with the information of the user to be created. 77

6.5 An example from the JSON schema used to generate the GUI for Airflow. The `radius` parameter is an integer parameter with a default value of 15. The `shape` parameter is a string parameter with two choices, `circular` and `rectangular`. The `enumNames` setting sets the display names for the enum values, while the enum values are the values sent to the workflow. The `source_file` parameter, is a string parameter that takes a file as input and encodes it as base64 before sending it to the workflow. 79
7.1 Parts of JSON schema describing the input parameters to generate the GUI. The `(...)` sections indicate that some of the code was left out for brevity. 93
7.2 The JSON schema for taking input for the RAM model. The dependencies specify when the input fields for RAM show up. If the `run_ram` parameter is false, nothing is showed, but if it is true, then all input fields specified in for RAM are shown. 94
7.3 JavaScript code implementing the sending of a request with parameters from the GUI application to the Airflow REST API. 95
7.4 The outputted format of the user input to the Airflow workflow. The data can be used without any changes as Airflow dag configuration. (...) indicates skipped values. 95
8.1 MATLAB `evalin` error that occurs when running the KNIME workflow. 101
8.2 MATLAB error about Java Virtual Machines that occurs when running the KNIME workflow. 101
8.3 MATLAB warning caused by the MATLAB Snippet nodes. 101
8.4 The Airflow error that occurred in one of the tasks. The error might be caused by a glibc bug [92]. 103
8.5 A section of a MATLAB script from the original package that loads the bathymetry. The `qnq` variable is fetched from the GUI and specifies how to color the map. 119
8.6 A section of the MATLAB function `load_bathymetry.m` that loads bathymetry in the workflow version of the package. 120

List of Figures

1.1 Illustration of generic sound speed profiles as a function of depth [22] 9
1.2 A workflow example for processing a sales dataset created in KNIME. The grey rectangular bar below the nodes indicates current execution status. 11
1.3 Design science research cycles by Hevner [29] 13

2.1 Ray paths computed with Bellhop for a ray simulation (a) and an eigenray simulation (b). The black lines hit the surface and the bottom and the green lines only hits the surface. Blue lines occur if only the bottom was hit, and red lines if neither the surface nor the bottom was hit. 17
2.2 Transmission loss computed with Bellhop as a function of range and depth for a source at 100 m radiating with 200 Hz frequency. (a) shows the coherent transmission loss, (b) the semi-coherent transmission loss, and (c) the incoherent transmission loss. 18
2.3 Results calculated with Eigenray for a source at 100 m depth and a receiver at 100 m depth. The top plot shows the arrival angle of the eigenrays. The bottom plot shows the path the eigenrays traveled from source to receiver in light blue. The darker blue line shows the bathymetry. 19
2.4 Results calculated with Eigenray timefronts for a source at 100 m depth and a receiver at 100 m depth. The top plot shows the turning point filter and the bottom plot shows the timefronts of the eigenrays. 19
2.5 Results calculated with the two different RAM models. The transmission loss versus range and depth is shown in (a) for a source at 100 m vibrating at 100 Hz. The turning-point filter and timefronts calculated with MPIRAM are shown in (b) for a source at 100 m vibrating at 50 Hz. 20
2.6 Plot of the KRAKEN model results. The top plot shows the turning point filter, the middle plot shows the timefronts, and the bottom plot shows the normal modes on top of the bathymetry (blue line) and sound speed profile (red line) plot. 21
2.7 Airflow DAG 23
2.8 Configuration view for a file reader node in KNIME. 25
2.9 A workflow example that takes input for and runs the RAM model. The light grey nodes are components that can be expanded to see the nodes inside. The darker grey node with a checkmark is a metanode. Metanodes can also be expanded to see the nodes within. 26

2.10 The expanded view of the `Run RAM model` node 26

2.11 KNIME menu to set up a user created component. The top input field specifies the name of the component, shown above it in the workflow. The left box contains the user specified input ports and what kind of input it accepts. The right port contains the user specified output ports. The current input and output ports take data in the form of a table, some of the other alternatives are image input and flow variable ports. 27

2.12 The Airflow webserver main page. Airflow contains one DAG, Arctic-Ocean, which has 23 successful runs and 19 failed. 29

2.13 Example of Airflow task compositions. (a) shows the Airflow DAG where each task depend on one previous task. (b) shows the Airflow DAG where a set of task depend on another set of tasks. 30

3.1 The main GUI in the MATLAB implementation of the Arctic Package [20] 36

3.2 The current structure of the Arctic Package. The white boxes represents a step executed in the Arctic Package. The purple boxes represent input parameters. The blue boxes represent database files used by the given step. The text in grey in the blue and purple boxes are files or parameters that are generated, but not used by the Arctic Package. 37

3.3 A map created with the Arctic Package. It uses the IBCAO database for the coloring of the area which shows the depth. The red areas represent the shallow regions, while blue represent the deeper regions. The green stars represent the sources, the magenta dots represent receivers, and the white lines represent the paths between the sources and the receivers. Some of the source and receiver coordinates are overlapping. 44

3.4 The bathymetry and sound speed profile on the transect from a source at 82.11° North and 26.32° East and a receiver at 83.45° North and 25.75° East. The sound speed profiles are the red lines, while the bathymetry is the blue line. 45

3.5 Plotting two model calculations on top of each other. (a) shows MPIRAM and timefronts calculated by Eigenray in the same plot. The top plot shows a turning point filter for MPIRAM in light blue and for Eigenray timefronts in red and black. The middle plot shows the timefronts for the eigenrays, calculated by Eigenray. (b) shows RAM and eigenrays calculated by Eigenray in the same plot. The transmission loss from RAM is plotted in blue and yellow, while the eigenray paths are plotted in pink. 45

4.1 The top level KNIME workflow that prepares model data, runs acoustic models, and plots modeling results. 49

4.2 The expanded view of the `Take input parameters` node in Figure 4.1. It has five sub-components that structure the input widgets into nodes that group the input parameters together based on which parameters belong to which tasks in the workflow. The component has two output ports, as the plotting of the map and the modeling uses different input parameters. 50
4.3 The expanded view of the `Take map input parameters` node in Figure 4.2. The node consists of widgets to take user input. The integer widget takes integer input and single selection widget creates a dropdown menu for the user. 51
4.4 The expanded map component shows the two steps needed to output a plot of the map. The first node is a MATLAB Snippet node and creates the map and the next node adds the image to a table. 52
4.5 The content of the `RAM` node in Figure 4.1 52
4.6 The expanded `Run RAM model` node from Figure 4.5 shows the steps needed to prepare data, run the RAM model, and plot the results. . . 54
4.7 The expanded Bellhop component shows the steps needed to prepare data, run the Bellhop model, and plot the results. 57
4.8 The expanded view of the KNIME map component which plots the map base, the sources, the receivers, and the paths between the sources and receivers. 60

5.1 The Airflow workflow that prepares model data, runs acoustic models, and plots modeling results. The `BranchPythonOperators` have a pink coloring, and the blue nodes are task groups. Task groups can be expanded to see the tasks within. The rest of the nodes are `BashOperators`. The green border indicates that the task was executed successfully. A red border would indicate a failed task and a pink border would indicate that a task was skipped. 62
5.2 The expanded view of the RAM task group from Figure 5.1 65

6.1 The KNIME tool view after successfully setting up the KNIME workflow. The file structure is shown in the left box named `KNIME Explorer`, the KNIME node options are shown in the `Node Repository` box, and the workflow is shown in the open tab: `ArcticOcean`. 74
6.2 The expanded view of the MPIRAM node in Figure 6.1 with the generated input view. The `Take MPIRAM input` node has been run, indicated by the green circle below the node, and is ready to take input. 75
6.3 An alternative way to open the input view in Figure 6.2. 76

7.1 The original GUI in the Arctic Package. It takes user input for creating the map (Figure 7.2) and for acoustic modeling. 81
7.2 The interactive map in the Arctic package. It is used to select sources and receivers and to view the coverage of different databases over a specified area. The intersection between the black lines, indicates the cursor placement. This is the same map as in Figure 3.3. 82
7.3 The GUI window for configuring the Eigenray parameters. When `GET PREDICT` is clicked, the user has to select a source and receiver from the map. 82

7.4 The advanced source and receiver GUI window. The user can manually add sources and receivers, nudge, remove, or save them to file. When a source or receiver is nudged, the user can move it from one placement on the map to another using the cursor. 83
7.5 The result plotter that allows the user to plot previously saved results to a figure. Some of the saved plots contain multiple different options for plotting that can be selected in the bottom drop-down. 83
7.6 The GUI generated by KNIME from widget nodes in Figure 7.7 specifying the map parameters. 84
7.7 The `Take map input parameters` node in the KNIME workflow. The node consists of widgets to take user input. The integer widget takes integer input and single selection widget creates a drop-down for the user. This figure is a copy of Figure 4.3. 85
7.8 The KNIME view to configure how the generated GUI is organized. The blue boxes represent the parameters, and where they are placed relative to each other in the GUI. For instance, the `Max latitude` parameter is located above the `Max longitude` parameter, as can be verified in Figure 7.6. 86
7.9 The full GUI generated by KNIME from widget nodes. The map section from Figure 7.6 is the middle section of the full GUI, starting from `Max latitude [°N]` to `Select the shape of the map`. 87
7.10 The top part of the GUI generated by RJSF for the Airflow workflow. The shown section takes the user input required for the map, in addition to source and receiver information. 89
7.11 The middle part of the GUI generated by RJSF for the Airflow workflow. The shown section takes the common parameters that all the acoustic models require for modeling. 90
7.12 The bottom part of the GUI generated by RJSF for the Airflow workflow. The shown section takes model input parameters. In this example, the RAM and Bellhop acoustic models are selected, and their specific parameters are set. The parameters for the other models are hidden. 91

8.1 The map plot in the new version of the Arctic Package. The white numbers identifies the sources and the magenta number identifies the receivers. A legend is also added to the plot to identify which points that are sources and which are receivers. 106
8.2 The GUI generated by KNIME. All the user input except for model specific parameters are set here. This figure is identical to Figure 7.9. 108
8.3 A model specific input view generated by KNIME. This figure is identical to Figure 7.3. 109
8.4 The top section of the Airflow GUI. The section shows the input fields related to the map. This figure is identical to Figure 7.10. 111
8.5 The bottom section of the Airflow GUI. This section has checkboxes to select which acoustic models to run, and input fields specific for the acoustic models selected. This figure is identical to Figure 7.12. . 113
8.6 Histogram of the complexity of the Arctic Package. 122

8.7 Histogram of the complexity of the Arctic Package normalized by total number of files. 122
8.8 RAM original result visualization (top) with temperature profiles and bathymetry (bottom) 127
8.9 RAM result visualization produced by the workflows. 127
8.10 Temperature profiles and bathymetry produced by the workflows. . . 127
8.11 (a) Shows the visualization of MPIRAM results in the original package, excluding the profile plot shown beneath the figures. (b) shows the visualization of MPIRAM results in the workflow version. 127
8.12 Memory and CPU usage on the PC used for testing when no other processes than measuring memory and CPU consumption were running.129
8.13 Memory usage (blue line) and CPU usage (red line) on the computer used for testing when the original Arctic Package was running. 130
8.14 Memory (blue line) and CPU usage (red line) on the computer used for testing when the KNIME workflow was running. The workflow was run three times, and then MATLAB was closed. 132
8.15 Memory usage (blue) and CPU usage (red) on the PC used for testing when the Airflow workflow was running. The workflow was run three times, and then the GUI, Airflow scheduler, and Airflow webserver was closed. The measurements were taken when the workflow was run with parallel execution enabled. 134
8.16 The memory usage of the three different versions of the Arctic Package. The blue line is the original MATLAB implementation, the green line is the KNIME workflow, and the orange line is the Airflow workflow. The execution times varies because these values also include the time spent to input values to the Arctic Package. 135
8.17 The CPU usage of the three different versions of the Arctic Package. The blue line is the original MATLAB implementation, the green line is the KNIME implementation, and the orange line is the Airflow implementation. This was measured at the same time as the memory usage in Figure 8.16. 135

9.1 The top level KNIME workflow model for the Arctic Package. This figure is a copy of Figure 4.1. 140
9.2 The constructed Airflow workflow model for the Arctic Package. This figure is a copy of Figure 5.1. 141

List of Tables

3.1 The analysis output of the source code created by CLOC [68] on the full Arctic Package. The analysis skipped 147 files that did not contain source code. Code lines excludes blank and comment lines, while total lines includes them. 39
3.2 The file structure in the Arctic Package. 40

4.1 The file structure of the MATLAB source code in the workflow version of the Arctic Package. 58
4.2 An analysis of the Arctic Ocean source code created by CLOC [68]. Code lines excludes blank and comment lines, while total lines includes them. The HTML and CSS files belong to the library `m_map`. 58

8.1 Evaluation criteria overview for the software quality models. The ISO/IEC criterion name functional suitability was shortened to functionality and performance efficiency was shortened to performance. . 96
8.2 Usability changes compared to the original GUI of the Arctic Package. A plus indicates an improvement for the criterion, while minus indicates that it has worsened. 104
8.3 Usability changes for the KNIME GUI compared to the original Arctic Package GUI. A plus indicates an improvement for the criterion, while minus indicates that it has worsened. 107
8.4 Usability improvements for the Airflow workflow GUI compared to the original GUI. A plus indicates an improvement for the criterion, while minus indicates that it has worsened. 110
8.5 Overview of which metrics affects which maintainability sub-criteria based on [93]. 116
8.6 Statistics showing how the MATLAB source code changed from the original Arctic Package to the new implementation. 117
8.7 User input parameters to the workflow for measuring performance . . 128
8.8 Execution times in the original workflow, rounded to one decimal . . 130
8.9 Execution times in the KNIME workflow, rounded to one decimal. . . 131
8.10 Execution times in the Airflow workflow, rounded to one decimal. . . 133
8.11 Evaluation summary and scoring. A score of 1 represents that the implementation poorly satisfies the criterion, while a score of 5 is a satisfactory criterion fulfillment. 136

9.1 Colored version of the evaluation summary and scoring table (Table 8.11). A score of 1 represents that the implementation poorly satisfies the criterion, while a score of 5 represents that the implementations satisfy the criterion. 142

Glossary

bathymetry The study of the seabed or other water body beds.

buoyancy The upwards force on an object in a fluid.

design cycle The design science research cycle that iterates between design, development, and evaluation of the artifact.

eigenray A ray that connects a source to a receiver.

enum A data type that consists of named values.

glider An unmanned underwater vehicle used to collect ocean data.

mooring A stationary collection of devices anchored to the sea floor.

Ocean Model The Arctic Package name for the ASTE dataset.

refraction Bending of a wave when the sound speed changes [21].

relevance cycle The design science research cycle that connects the project to the environment and aims to improve artifact according to the stakeholder needs.

rigor cycle The design science research cycle that provides past knowledge to the research project and that adds knowledge to the existing knowledge base from the project.

sound speed profile field A field of sound speed profiles c(z), over a range c(r,z).

Acronyms

ASTE Arctic Subpolar gyre sTate Estimate.

CAATEX The Coordinated Arctic Acoustic Thermometry Experiment.

CD Continuous Delivery/Deployment.

CI Continuous Integration.

CRUD Create, Read, Update, and Delete.

DAG Directed Acyclic Graph.

DevOps Development and Operations.

DSC Deep Sound Channel.

DSR Design Science Research.

ECCO Estimating the Circulation and Climate of the Ocean.

GUI Graphical User Interface.

HYCOM Hybrid Coordinate Ocean Model.

IBCAO International Bathymetric Chart of the Arctic Ocean.

LoC Lines of Code.

MEX Matlab EXecutable.

MPI Message Passing Interface.

NERSC Nansen Environmental Remote Sensing Center.

PE Parabolic Equation.

RAM Range Dependent Acoustic Model.

RJSF React-JSONSchema-Form.

WOA World Ocean Atlas.

WSL Windows Subsystem for Linux.

XComs Cross communication.

Bibliography

[1] *Caatex: Coordinated arctic acoustic thermometry experiment*. [Online]. Available: https://www.nersc.no/project/caatex (visited on Sep. 10, 2020).

[2] *Nansen environmental and remote sensing center: About us*. [Online]. Available: https://www.nersc.no/about (visited on Oct. 11, 2020).

[3] *Acoustic and oceanography*. [Online]. Available: https://www.nersc.no/group/acoustic-and-oceanography (visited on Apr. 12, 2021).

[4] W. Munk, *Ocean acoustic tomography*, eng, Cambridge, 1995.

[5] B. Dushaw, "Acoustic tomography, ocean," in *Encyclopedia of Remote Sensing*, E. G. Njoku, Ed. New York, NY: Springer New York, 2014, pp. 4–11, ISBN: 978-0-387-36699-9. DOI: 10.1007/978-0-387-36699-9_211.

[6] T. Duda, J. Bonnel, E. Coelho, and K. D. Heaney, "Computational acoustics in oceanography: The research roles of sound field simulations," *Acoustics Today*, vol. 15, p. 28, Jan. 2019. DOI: 10.1121/AT.2019.15.3.28.

[7] P. C. Etter, *Underwater acoustic modeling and simulation*, eng, Boca Raton, Florida, 2018.

[8] G. Hope, H. Sagen, E. Storheim, H. Hobæk, and L. Freitag, "Measured and modeled acoustic propagation underneath the rough arctic sea-ice," eng, *The Journal of the Acoustical Society of America*, vol. 142, no. 3, pp. 1619–1633, 2017. DOI: 10.1121/1.5003786.

[9] A. D. Hawkins and A. N. Popper, "A sound approach to assessing the impact of underwater noise on marine fishes and invertebrates," *ICES Journal of Marine Science*, vol. 74, no. 3, pp. 635–651, Dec. 2016. DOI: 10.1093/icesjms/fsw205. eprint: https://academic.oup.com/icesjms/article-pdf/74/3/635/31243362/fsw205.pdf.

[10] M. S. Ballard, "Three-dimensional acoustic propagation effects induced by the sea ice canopy," eng, *The Journal of the Acoustical Society of America*, vol. 146, no. 4, EL364–EL368, 2019. DOI: 10.1121/1.5129554.

[11] Y.-T. Lin, A. E. Newhall, J. H. Miller, G. R. Potty, and K. J. Vigness-Raposa, "A three-dimensional underwater sound propagation model for offshore wind farm noise prediction," eng, *The Journal of the Acoustical Society of America*, vol. 145, no. 5, EL335–EL340, 2019. DOI: 10.1121/1.5099560.

[12] Y.-T. Lin, M. B. Porter, F. Sturm, M. J. Isakson, and C.-S. Chiu, "Introduction to the special issue on three-dimensional underwater acoustics," eng, *The Journal of the Acoustical Society of America*, vol. 146, no. 3, pp. 1855–1857, 2019. DOI: 10.1121/1.5126013.

[13] B. Dushaw, *Arctic package*, Not available online, contact admin@nersc.no, 2018.

[14] M. Porter, "The bellhop manual and user's guide: Preliminary draft," Jan. 2011.

[15] B. D. Dushaw and J. A. Colosi, *Ray Tracing for Ocean Acoustic Tomography.* 1998.

[16] M. B. Porter, *The KRAKEN Normal Mode Program.* 1992.

[17] B. Dushaw, *Mpiram: The ram parabolic equation code in fortran 95*, 2015. [Online]. Available: `http://staff.washington.edu/dushaw/AcousticsCode/RamFortranCode.html` (visited on Dec. 8, 2020).

[18] M. D. Collins, *Users guide for ram versions 1.0 and 1.0p*, Naval Research Laboratory, 1999.

[19] E. Van den Bergh, "Further development of a software for acoustic propagation modeling," Master's thesis, University of Bergen, Western Norway University of Applied Sciences, 2020.

[20] F. T. Klockmann, "Optimization of software for modeling acoustic monitoring systems," Master's thesis, University of Bergen, Western Norway University of Applied Sciences, 2020.

[21] U. of Rhode Island and I. S. Center, *How does sea ice affect how sound travels?* [Online]. Available: `https://dosits.org/science/movement/sea-ice/` (visited on Jan. 12, 2021).

[22] F. B. Jensen, W. A. Kuperman, M. B. Porter, and H. Schmidt, *Computational Ocean Acoustics.* Springer New York, 2011, ISBN: 978-1-4419-8677-1. DOI: `10.1007/978-1-4419-8678-8`.

[23] M. Jakobsson *et al.*, "The international bathymetric chart of the arctic ocean (IBCAO) version 3.0," *Geophysical Research Letters*, vol. 39, no. 12, Jun. 2012. DOI: `10.1029/2012gl052219`.

[24] B. Ludäscher, S. Bowers, and T. McPhillips, "Scientific workflows," in *Encyclopedia of Database Systems*, L. LIU and M. T. ÖZSU, Eds. Boston, MA: Springer US, 2009, pp. 2507–2511, ISBN: 978-0-387-39940-9. DOI: `10.1007/978-0-387-39940-9_1471`.

[25] C. Liew, M. Atkinson, M. Galea, T. Ang, P. Martin, and J. Hemert, "Scientific workflows: Moving across paradigms," eng, *ACM computing surveys*, vol. 49, no. 4, pp. 1–39, 2017. DOI: `10.1145/3012429`.

[26] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," eng, *IEEE software*, vol. 33, no. 3, pp. 94–100, 2016. DOI: `10.1109/MS.2016.68`.

[27] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*, eng, 2014th ed. Berlin, Heidelberg: Springer Berlin Heidelberg, ISBN: 3662438380. DOI: `10.1007/978-3-662-43839-8`.

[28] P. Johannesson and E. Perjons, *An Introduction to Design Science*, eng, 2014th ed. Cham: Springer International Publishing AG, 2014, ISBN: 3319106317. DOI: `10.1007/978-3-319-10632-8`.

[29] A. Hevner, "A three cycle view of design science research," *Scandinavian Journal of Information Systems*, vol. 19, Jan. 2007.

[30] J. A. Colosi, *Sound Propagation through the Stochastic Ocean*. Cambridge University Press, 2016. DOI: 10.1017/CBO9781139680417.

[31] J. Bowlin, J. Spiesberger, T. Duda, and L. Freitag, "Ocean acoustical ray-tracing software ray," Oct. 1993. DOI: 10.1575/1912/618.

[32] M. D. Collins, "A split-step padé solution for the parabolic equation method," *The Journal of the Acoustical Society of America*, vol. 93, no. 4, pp. 1736–1742, Apr. 1993. DOI: 10.1121/1.406739. [Online]. Available: 10.1121/1.406739.

[33] D. Gannon, E. Deelman, M. Shields, and I. Taylor, "Control- versus data-driven workflows," in *Workflows for e-Science: Scientific Workflows for Grids*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. London: Springer London, 2007, pp. 1–8, ISBN: 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_1.

[34] M. Vigder, N. Vinson, J. Singer, D. Stewart, and K. Mews, "Supporting scientists' everyday work: Automating scientific workflows," *IEEE software*, vol. 25, no. 4, pp. 52–58, 2008. DOI: 10.1109/MS.2008.97.

[35] N. Radosevic, M. Duckham, G.-J. Liu, and Q. Sun, "Solar radiation modeling with knime and solar analyst: Increasing environmental model reproducibility using scientific workflows," *Environmental modelling software : with environment data news*, vol. 132, 2020. DOI: 10.1016/j.envsoft.2020.104780.

[36] M. Shields, "Control- versus data-driven workflows," in *Workflows for e-Science: Scientific Workflows for Grids*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. London: Springer London, 2007, pp. 167–173, ISBN: 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_11.

[37] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," eng, *Journal of grid computing*, vol. 13, no. 4, pp. 457–493, 2015. DOI: 10.1007/s10723-015-9329-8.

[38] V. Curcin and M. Ghanem, "Scientific Workflow Systems - Can One Size Fit All?" *Cairo International Biomedical Engineering Conference*, 2008. DOI: 10.1109/CIBEC.2008.4786077.

[39] E. Elmroth, F. Hernández, and J. Tordsson, "Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment," eng, *Future generation computer systems*, vol. 26, no. 2, pp. 245–256, 2010. DOI: 10.1016/j.future.2009.08.011.

[40] B. Ludäscher *et al.*, "Scientific workflow management and the kepler system," eng, *Concurrency and Computation: Practice and Experience*, vol. 12, no. 15, pp. 1039–1065, 2006, ISSN: 1096-9128. DOI: 10.1002/cpe.994.

[41] E. Deelman *et al.*, "Pegasus, a workflow management system for science automation," eng, *Future generation computer systems*, vol. 46, no. C, pp. 17–35, 2015. DOI: 10.1016/j.future.2014.10.008.

[42] K. Wolstencroft *et al.*, "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud," *Nucleic Acids Research*, vol. 41, pp. 557–561, W1 2013. DOI: 10.1093/nar/gkt328.

[43] S. Marru *et al.*, “Apache airavata: A framework for distributed applications and computational workflows,” in *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, ser. GCE ’11, Seattle, Washington, USA: Association for Computing Machinery, 2011, pp. 21–28, ISBN: 9781450311236. DOI: `10.1145/2110486.2110490`.

[44] Apache Software Foundation, *Airflow*, version 1.10.13, https://airflow.apache.org/, Aug. 17, 2020.

[45] Prefect, *Prefectcore*, version 0.14.12, https://www.prefect.io, Mar. 10, 2021.

[46] E. Afgan *et al.*, “The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update,” eng, *Nucleic acids research*, vol. 46, no. W1, W537–W544, May 2018, ISSN: 0305-1048. DOI: `10.1093/nar/gky379`.

[47] M. R. Berthold *et al.*, “KNIME: The Konstanz Information Miner,” in *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*, Springer, 2007, ISBN: 978-3-540-78239-1.

[48] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher, “Scientific workflow design for mere mortals,” eng, *Future generation computer systems*, vol. 25, no. 5, pp. 541–551, 2009. DOI: `10.1016/j.future.2008.06.013`.

[49] KNIME AG, *Knime analytics platform*, version 4.3, https://www.knime.com/, Jun. 12, 2020.

[50] A. Berlin-Taylor, *Apache airflow 2.0 is here!* Dec. 2020. [Online]. Available: `https://airflow.apache.org/blog/airflow-two-point-oh-is-here/` (visited on May 26, 2021).

[51] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[52] M. Stöter, A. Niederlein, R. Barsacchi, F. Meyenhofer, H. Brandl, and M. Bickle, “Cellprofiler and knime: Open source tools for high content screening,” English, in *Target Identification and Validation in Drug Discovery*, ser. Methods in Molecular Biology, J. Moll and R. Colombo, Eds., vol. 986, Humana Press, 2013, pp. 105–122, ISBN: 978-1-62703-310-7. DOI: `10.1007/978-1-62703-311-4_8`. [Online]. Available: `http://dx.doi.org/10.1007/978-1-62703-311-4_8`.

[53] S. Beisken, T. Meinl, B. Wiswedel, L. F. de Figueiredo, M. Berthold, and C. Steinbeck, “Knime-cdk: Workflow-driven cheminformatics,” eng, *BMC bioinformatics*, vol. 14, no. 1, pp. 257–257, 2013. DOI: `10.1186/1471-2105-14-257`.

[54] C. Steinbeck, C. Hoppe, S. Kuhn, M. Floris, R. Guha, and E. Willighagen, “Recent developments of the chemistry development kit (CDK) - an open-source java library for chemo- and bioinformatics,” *Current Pharmaceutical Design*, vol. 12, no. 17, pp. 2111–2120, Jun. 2006. DOI: `10.2174/138161206777585274`.

[55] C. T. Rueden *et al.*, “ImageJ2: ImageJ for the next generation of scientific image data,” *BMC Bioinformatics*, vol. 18, no. 1, Nov. 2017. DOI: `10.1186/s12859-017-1934-z`.

[56] A. Kooistra *et al.*, “3d-e-chem: Structural cheminformatics workflows for computer-aided drug discovery,” eng, *ChemMedChem*, vol. 13, no. 6, pp. 614–626, 2018. DOI: `10.1002/cmdc.201700754`.

[57] J. Gally, S. Bourg, Q. Do, S. Aci-Sèche, and P. Bonnet, "Vsprep: A general knime workflow for the preparation of molecules for virtual screening," eng, *Molecular informatics*, vol. 36, no. 10, 1 700 023–n/a, 2017. DOI: 10.1002/minf.201700023.

[58] A. Tuerkova and B. Zdrazil, "A ligand-based computational drug repurposing pipeline using knime and programmatic data access: Case studies for rare diseases and covid-19," eng, *Journal of cheminformatics*, vol. 12, no. 1, pp. 71–71, 2020. DOI: 10.1186/s13321-020-00474-z.

[59] G. Nicola, M. R. Berthold, M. P. Hedrick, and M. K. Gilson, "Connecting proteins with drug-like compounds: Open source drug discovery workflows with bindingdb and knime," eng, *Database : the journal of biological databases and curation*, vol. 2015, bav087, 2015. DOI: 10.1093/database/bav087.

[60] H. Strobelt *et al.*, "Hitsee knime: A visualization tool for hit selection and analysis in high-throughput screening experiments for the knime platform," eng, *BMC bioinformatics*, vol. 13 Suppl 8, no. S8, S4–S4, 2012, ISSN: 1471-2105.

[61] A. J. Jara, D. Genoud, and Y. Bocchi, "Big data for smart cities with knime a real experience in the smartsantander testbed," eng, *Software, practice experience*, vol. 45, no. 8, pp. 1145–1160, 2015, ISSN: 0038-0644.

[62] X. Li, J. Song, and R. Huang, "A kepler scientific workflow to facilitate and standardize marine monitoring sensor parsing and dynamic adaption," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*, IEEE, 2014, pp. 1023–1026, ISBN: 978-1-4799-3279-5. DOI: 10.1109/ICSESS.2014.6933739.

[63] M. H. Nguyen, D. Crawl, T. Masoumi, and I. Altintas, "Integrated machine learning in the kepler scientific workflow system," eng, *Procedia computer science*, vol. 80, pp. 2443–2448, 2016. DOI: 10.1016/j.procs.2016.05.545.

[64] D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis," in *Workflows for e-Science: Scientific Workflows for Grids*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. London: Springer London, 2007, pp. 39–59, ISBN: 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_4.

[65] D. Dossett and M. Sevior, "Prompt calibration automation at belle ii," eng, *EPJ Web of conferences*, vol. 245, p. 02 016, 2020. DOI: 10.1051/epjconf/202024502016.

[66] E. Antonov, B. Onykiy, A. Artamonov, and E. Tretyakov, "Information analysis support for decision-making in scientific and technological development," *International Journal of Technology*, vol. 11, no. 6, pp. 1125–1135, Dec. 2020. DOI: 10.14716/ijtech.v11i6.4465.

[67] S. Callaghan *et al.*, "Metrics for heterogeneous scientific workflows: A case study of an earthquake science application," eng, *The international journal of high performance computing applications*, vol. 25, no. 3, pp. 274–285, 2011. DOI: 10.1177/1094342011414743.

[68] A. Danial, *Cloc - count lines of code*, version 1.88, https://github.com/AlDanial/cloc, Apr. 15, 2021.

[69] R. Pawlowicz, *M_map: A mapping package for matlab*, version 1.4m, 2020. [Online]. Available: `www.eoas.ubc.ca/~rich/map.html`.

[70] G. Forget, J.-M. Campin, P. Heimbach, C. N. Hill, R. M. Ponte, and C. Wunsch, "Ecco version 4: An integrated framework for non-linear inverse modeling and global ocean state estimation," eng, *Geoscientific model development*, vol. 8, no. 10, pp. 3071–3104, 2015. DOI: `10.5194/gmd-8-3071-2015`.

[71] T. P. Boyer *et al.*, *World Ocean Atlas 2018*, en, Dataset, National Centers for Environmental and Information (NCEI). [Online]. Available: `https://www.ncei.noaa.gov/access/metadata/landing-page/bin/iso?id=gov.noaa.nodc:NCEI-WOA18` (visited on Mar. 5, 2021).

[72] A. Köhl, D. Stammer, and B. Cornuelle, "Interannual to decadal changes in the ecco global synthesis," eng, *Journal of physical oceanography*, vol. 37, no. 2, pp. 313–337, 2007. DOI: `10.1175/JPO3014.1`.

[73] A. Nguyen *et al.*, "On the benefit of current and future alps data for improving arctic coupled ocean-sea ice state estimation," eng, *Oceanography (Washington, D.C.)*, vol. 30, no. 2, pp. 69–73, 2017. DOI: `10.5670/oceanog.2017.223`.

[74] *Acoustic models*, contact eirin.sognnes@outlook.com for access. [Online]. Available: `https://github.com/EirinS/AcousticModels` (visited on Apr. 26, 2021).

[75] *Arctic.ocean-v2*, contact eirin.sognnes@outlook.com for access. [Online]. Available: `https://github.com/EirinS/Arctic.Ocean-v2` (visited on Apr. 26, 2021).

[76] E. Sognnes, *Knime-workflow*. [Online]. Available: `https://github.com/EirinS/KNIME-workflow` (visited on Apr. 26, 2021).

[77] E. Sognnes, *Airflow arctic ocean workflow*. [Online]. Available: `https://github.com/EirinS/Airflow-workflow` (visited on Apr. 26, 2021).

[78] Python Software Foundation, *Python*, version 3.7, https://www.python.org/, Jun. 27, 2018.

[79] I. Docker, *Docker*, version 20.10.5, https://www.docker.com/, Mar. 2, 2021.

[80] MATLAB, *R2020b*, Natick, Massachusetts: The MathWorks Inc., 2020.

[81] S. D. Roughley, "Five years of the KNIME vernalis cheminformatics community contribution," *Current Medicinal Chemistry*, vol. 27, no. 38, pp. 6495–6522, Nov. 2020. DOI: `10.2174/0929867325666180904113616`.

[82] OpenJS Foundation, *Node.js*, version 15.2.0, https://nodejs.org/, Nov. 10, 2020.

[83] *React-jsonschema-form*, version 2.4.2, Feb. 2, 2021. [Online]. Available: `https://react-jsonschema-form.readthedocs.io`.

[84] S. kalyan Sozhavaram, *Enable cors (cross-origin request sharing) for the api*, Jun. 2019. [Online]. Available: `https://issues.apache.org/jira/browse/AIRFLOW-4804` (visited on Apr. 27, 2021).

[85] ISO/IEC, "Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," eng, Standard ISO/IEC 25010:2011, 2011. [Online]. Available: `https://www.iso.org/standard/35733.html`.

[86] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality," en, GENERAL ELECTRIC CO SUNNYVALE CA, Tech. Rep., Nov. 1977. [Online]. Available: `https://apps.dtic.mil/sti/citations/ADA049014` (visited on Mar. 28, 2021).

[87] R. Dromey, "A model for software product quality," eng, *IEEE transactions on software engineering*, vol. 21, no. 2, pp. 146–162, 1995. DOI: `10.1109/32.345830`.

[88] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs: Prentice-Hall, 1992, ISBN: 0137203845.

[89] I. Ozkaya, L. Bass, R. Nord, and R. Sangwan, "Making practical use of quality attribute information," eng, *IEEE software*, vol. 25, no. 2, pp. 25–33, 2008. DOI: `10.1109/MS.2008.39`.

[90] D. Galin, *Software quality : Concepts and practice*, eng, Hoboken, New Jersey, 2017.

[91] J. Estdale and E. Georgiadou, "Applying the iso/iec 25010 quality models to software product," in *Systems, Software and Services Process Improvement*, X. Larrucea, I. Santamaria, R. V. O'Connor, and R. Messnarz, Eds., Cham: Springer International Publishing, 2018, pp. 492–503, ISBN: 978-3-319-97925-0. DOI: `10.1007/978-3-319-97925-0_42`.

[92] S. Nagy, *Bug 19329 - dl-tls.c assert failure at concurrent* $pthread_{c}reateanddlopen$. [Online]. Available: `https://sourceware.org/bugzilla/show_bug.cgi?id=19329` (visited on Apr. 16, 2021).

[93] R. Baggen *et al.*, "Standardized code quality benchmarking for improving software maintainability," eng, *Software quality journal*, vol. 20, no. 2, pp. 287–307, 2012. DOI: `10.1007/s11219-011-9144-9`.

[94] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," eng, in *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, IEEE, 2007, pp. 30–39, ISBN: 0769529488. DOI: `10.1109/QUATIC.2007.8`.

[95] PMD, *Pmd*, version 6.33.0, https://pmd.github.io/, Mar. 27, 2021.

[96] T. McCabe, "A complexity measure," eng, *IEEE transactions on software engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. DOI: `10.1109/TSE.1976.233837`.

Appendix A

MATLAB Log

```
    MATLAB Startup Performance Metrics (In Seconds)

total   item     gap      description
=========================================================
 0.00   0.00    0.00   MATLAB script
 0.05   0.05    0.00   main
 0.48   0.42   -0.00   Session Initialize
 0.52   0.00    0.04   Toolbox cache load Start
 0.60   0.08    0.00   LM Startup
 0.69   0.00    0.09   splash
 0.83   0.30    0.53   cachepath
 1.06   0.18    0.20     Constant Initialization
 1.08   0.39    0.00   Engine Startup
 1.29   0.21    0.00      InitSunVM
 3.34   1.91    0.14      PostVMInit
 3.34   2.26    0.00     mljInit
 3.35   2.26    0.00   Java initialization
 3.44   0.00    0.09   psParser
 3.44   0.00    0.00   Toolbox cache join
 3.79   0.25    0.10   matlabpath
 3.79   0.00    0.00   MATLAB Addons registration
 3.89   0.00    0.10   Registration Framework Initializer Start
 3.90   0.01    0.00   matlabrc
=========================================================
Items shown account for 96.6% of total startup time [TIMER: 10 MHz]
```