Climate Models: A Software Engineering Approach

Fernando G. Tinetti[*], Mariano Méndez

Technical Report LS-01-2014
July 2014

III-LIDI, Facultad de Informática, UNLP
50 y 120, 1900, La Plata
Argentina

Abstract

Climate Simulation and Weather Forecasting are amongst the most representative examples of scientific software, which has evolved throughout the past sixty years. In this paper, a set of Global Climate Models (GCM) have been analysed from a Software Engineering perspective, analysing the composition of their internal structure and programming constructs which have been used in the building process. We have implemented a set of software metrics such as Cyclomatic Complexity, Lines of Code, Number of Fortran Obsolete Language Features, among others.We have followed a compiler like approach, collecting information based on traversing the Abstract Syntax Tree (AST). The obtained data can be used for different purposes at different stages of the software life cycle such as: maintenance tasks, parallelization, and optimization. The results suggest that some programming techniques used for building scientific software have fallen into disuse because they are now considered obsolete and error-prone. In addition, GCM's internal structure seems to evolve at a slower pace than programming techniques. The analysis methodology can be used to update and enhance the scientific software in order to make simpler other tasks such as optimization and parallelization for specific new hardware such as multi/many-core processors and co-processors, distributed memory parallel hardware, etc.

1 Introduction

In the early beginnings of Computer Science, the production of software was limited to scientific software. Some of the most preponderant examples of such software are Weather Forecasting and Climate Simulations. The first program which produced weather forecast prediction run on a computer in 1950 [21, 69, 41], and area of scientific computing, or rather, the area of numerical processing, has

[*]also at Comisión de Investigaciones Científicas de la Prov. de Bs. As.

been producing software for many decades. These programs are models which describe complex phenomena from reality based on a series of mathematical equations which are solved by an algorithm which is executed as a computer code [53]. The year 1954 saw the birth of the most long-lived programming language [49], and by far the most used in scientific software production and High Performance Computing (HPC): Fortran [12, 55, 52, 7].

Some research conducted recently has stressed an apparent divergence between the techniques and methods adopted in commercial software production and those used for producing scientific software [70, 36, 57, 8, 12, 58, 13]. We have measured some source code characteristics to determine how many changes on source code should be applied as well as which kind of program transformations should be. Also, those measurements should be useful to obtain further information that would help us to quantify the changes needed. We have focused our analysis on a set of scientific applications in the field of climatology. We have worked on Global Climate Models (GCM) by performing a detailed source code measurements, and "Our purpose has been purely scientific in an attempt to find how things are, without moralizing or judging people's competence" [38] or their work.

If we determine that the software we are working with needs be to modernized or parallelized, we should, as an initial step, break through from the "Unknown" to a "Manageable" process/project. There are some issues that generally affect Fortran source code, particularly in legacy Fortran [66] program, e.g. software seems to have been built with old design styles, documentation is rarely found, different programming styles are found in source code lines, and the wide usage of Fortran Common Blocks for sharing memory among subroutines. We are working on legacy scientific Fortran source code from many points of view, including source code transformation/s for HPC [67], and also gathering information from researchers, Fortran software developers, Fortran standard meeting group members, etc. as indicated in the poll at [73].

Fortran has faced a particular evolution throughout its own life; during the last decades the Fortran standard has been revised several times (1978, 1991, 1997, 2004, 2010) [49]. Modernizing and updating old Fortran source code has become an important objective because a large set of programs written over more than forty years are still operational these days. Scientific software can be considered as a good case study.

A major common problem that scientists are faced with is the need for better performance. One reliable option for such aim is to take advantage of parallel processing by using shared memory parallel hardware. Even though multi-core processors are widely spread, even in small scale computers, their facilities are not fully exploited. When it comes to taking advantage of shared memory parallel processing a good starting point is using libraries or facilities like those provided by threading. However, if we venture into the parallelization task, we will find some obstacles closely related to the aforementioned modernization issues. Nevertheless, there are Fortran programs which present a large set of old language features such as: fixed format without any indentation, obsolete language features still in use, deleted language features which are allowed by compilers, and so forth. These features make the parallelization process a very arduous and error prone task.

This study performs a thorough analysis on a set of scientific programs commonly known as Climate Models in order to find traces of differences between scientific and commercial software reflected in the source code. Once found, we should determine the way in which the Fortran language is used to build this kind of software. In addition, we should also determine whether or not a specific set of source code transformations should be applied to the source code in order to help it evolve, modernize, optimize, parallelize, etc.

This paper is structured as follows. The next section makes a historical review about forecasting and climate models including those climate models selected to be analysed. In section 3 we describe current problems found in scientific software from a software engineering point of view. In Section 4 the methodology adopted to perform our research is thoroughly described. Results are presented in Section 5. Section 6 is devoted to the discussion and further comments about the results presented in section 5. A brief literature review is made in Section 7. Finally, Section 8 presents conclusions and further work.

2 Climate Models and Weather Forecast: An Historical Review

The birth of modern meteorology and weather forecasting can be traced back to the beginning of the 20th century [21, 69, 41]. In 1904, Vilhem Bjerknes published an article in which he proposed a procedure and a set of equations that allowed scientists to forecast weather. This set of seven independent equations described the behaviour of seven variables: temperature, pressure, density, the three components of velocity, and humidity. Furthermore, he proposed a two step process for obtaining a rational forecast: the diagnostic step and the prognostic step. The solutions to Bjerknes equations required a large amount of mathematical calculations to be performed; given the lack of automated tools to achieve those calculations, the success of the proposed model was hindered.

In the early '20s, the idea of dividing space into a grid composed by cells, each one with its own value for different variables (temperature, pressure, density, the three components of velocity and humidity) and the applying of the finite difference method to solve differential equations, converged into the first "Numerical Weather Prediction" authored by L. F. Richardson. While Richardson's techniques made use of a simplified set of Bjerkenes equations the computations required to perform the calculations were still very time consuming. In order to speed up the process, Richardson imagined what he named "the forecast factory" an avant-garde version of automated collaborative calculation process. It would take a crowd of 64000 people, each of them equipped with a mechanical calculator, to execute a small part of the whole calculation, an early vision of crowd computing [21, 69].

After the World War II, John Von Neumann, who had been embarked on computational simulations of nuclear weapon explosions, related these two non-linear problems of fluid dynamics. In the early '50s Von Neumann gathered a group of climate scientists, thus founding the Meteorology Project. Von Neumann promoted the development of weather modelling: he "hoped that weather modelling might lead to weather control" [21]. This group of scientists lead by Jule Charney, run the first automated regional weather forecast on a computer in 1950; the model run on the Electronic Numerical Integrator And Computer (ENIAC) [21, 69, 41].

This simulation split North America into a 270 cell points grid, each cell point separated by 270 Km. The next step was building a model of the Earth's atmosphere. The first General Circulation Model was run on May 1955 by Norman Phillips. This was a two level quasi-geotrophic model, it was the first large-scale atmosphere simulation to run on a computer, the model used a grid of 272 cells [41], and it was ran on the Mathematical Analyzer, Numerical Integrator, and Computer (MANIAC I). The next big step in this evolutionary process was the addition of interactions between the atmosphere and other Earth components such as the ocean. When a general circulation model is coupled with an ocean model, an Ocean-Atmosphere General Circulation Model (OAGCM) is obtained. The first attempt of OAGCM was presented by Manabe and Bryan in 1969. In 1975 they presented an improved model that obtained more accurate results [43, 21]. From 1975 to 1985 more sophisticated techniques contributed to the integration of the atmosphere with other climate factors such as ice sea, land/vegetation, and carbon cycle among others.

Nowadays an OAGCM has evolved from a set of seven independent equations into a complex structure of constituents of interacting components. Some of these components are used modern climate models, such as:

- The Atmosphere Component, to model physical properties and the dynamics of the atmosphere.
- The Ocean Component, to model physical properties and the dynamics of the ocean.
- The Land Component, to model physical properties and the dynamics of the land surface.
- The Sea Ice Component, to model physical aspects of the Sea Ice such as thermodynamical properties and the ice dynamics.
- The coupler, to transfer values of state variables (energy, heat, water) between model components.

Thus, climate models have evolved into complex entities composed by a set of equations as well as people, physical, and mathematical models, equations, software tools, and hardware to name just a few. The combination of all of these resources allows meteorologists to obtain, predict and validate the large-scale atmosphere dynamics. The underlying idea in paper is that a complex program such as a climate model should be adecuately constructed and maintained, and cannot be left "as is" just because of its complexity. Furthermore, considering a climate model (or any other numerical/processing program, actually) as "untouchable" implies lots of disadvantages from the point of view of performance improvements relative to both current and new hardware facilities/platforms. In the rest of the article we will use GCM to refer to both "General Climate Models" and "Global Climates Models" which are considered "the most sophisticated climate models" [54].

2.1 Climate Models and Climate Change

Driven by climate change, the WCRP (World Climate Research Program) Working Group on Coupled Modelling (WGCM), promoted a new set of coordinated climate model experiments in the year 2008. These experiments have been performed by 20 climate modeling groups from around the world and integrate the fifth phase of the Coupled Model Intercomparison Project (CMIP5), which will aid in [65]:

1. Determining the mechanisms behind model differences in not very well understood feedbacks connected to the carbon cycle and clouds.
2. Evaluating climate predictability and examining the models' ability to predict climate on decadal time scales.
3. Determining the reasons why analogously forced models generate a given range of responses.

All of the involved models are computing intensive since a climate simulation for a period of one hundred years can be very time consuming even on large and powerful modern supercomputers with thousands of processors [68].

The original aim of our research was to analyse the whole set of climate models belonging to the participating groups of the CMIP5 experiments. We requested the collaboration of each modelling group in order to obtain each model's source code. In response to our request, we received the climate models specified in Table 1.

Modelling Group	Model	Institution
BCC	BCC-CSM1.1	Beijing Climate Center, China Meteorological Administration
CSIRO-QCCCE	CSIRO-Mk3.6.0	Commonwealth Scientific and Industrial Research Org. in collaboration with the Queensland Climate Change Centre of Excellence
INM	INM-CM4	Institute for Numerical Mathematics
IPSL	IPSL	Institut Pierre-Simon Laplace
MOHC	HadGEM2 HadGEM3	Met Office Hadley Centre
MPI-M	MPI-ESM-LR	Max Planck Institute for Meteorology (MPI-M)
NASA GISS	ModelE GISS	NASA Goddard Institute for Space Studies
NASA GMAO	GEOS-5	NASA Global Modeling and Assimilation Office
NCAR	CCSM3 CCSM4	National Center for Atmospheric Research
NOAA GFDL	GFDL-CM2.1	Geophysical Fluid Dynamics Laboratory
NSF-DOE-NCAR	CESM1	National Science Foundation, Department of Energy, National Center for Atmospheric Research
CMCC-CESM	CMCC-CESM	Centro Euro-Mediterraneo per I Cambiamenti Climatici
COLA and NCEP	CFSv2-2011	Center for Ocean-Land-Atmosphere Studies and National Centers for Environmental Prediction

Table 1: Analysed Climate Models and Modelling Groups

3 Software Engineering and Scientific Computing

In its early years the scope of computing was limited to scientific computing, i.e. "computing" was regarded as a synonym of "scientific computing" [35]. People outside academic scientific areas did not have access to a real computer. In 1959 "Computer Science" was used for the first time [24] and in just a few decades it has had wide and startling influence on human life. Since then, Computer Science has evolved to include a vast number of subfields including Software Engineering [59].

In comparison, the evolution of scientific computing seems to be taking place at a much slower rate. Moreover, not only it does seem to be lagging behind, but also it seems to have taken on a different direction altogether. Some authors refer to this phenomenon a the "mismatch" or "chasm" between Software Engineering and Scientific Computing [35, 8, 20]. The current software building practices used by computational scientists often bear little resemblance to those promoted by Software Engineers [8, 58, 12, 29, 20, 35]. A possible reason could be that it is commonly found that the role of the end-user and programmer fall upon the same person [58, 36, 8, 70, 12, 29, 35, 57]. Even when hardware and computational power is growing at an accelerated rate, computational science seems to be lagging behind [70, 20].

Among some of the alleged reasons for the aforementioned mismatch in software development, we could start by highlighting the complexity of the domain [58, 57]. A second factor could be the low value scientists ascribe to software developing knowledge and skills [8, 58, 12]. Third, lack of software engineering training and the horizontal quality of teaching, where knowledge is passed on from scientist to scientist both subject having the same level of programming competence [70, 8]. Also, scientists are reticent to use and apply modern programming tools such as integrated development environments (IDE) [70, 8, 12, 13]. In spite of the existing mismatch between software engineering and scientific compunting, the programs created by scientists are successful [56].

3.1 Scientific Computing and Fortran

One of the most widely used programming languages for developing scientific and computing intensive software is Fortran [8, 36, 12, 13, 55]. We could start by describing Fortran as a long-lived programming language given the fact that it was born in 1954. The language reference book was first published in 1957 and it featured only 32 statements [6]. While many people may consider Fortran as an obsolete, ancient, out-of-date, and irrelevant programming language to be taught by the computing departments of a large number of universities [36], it is in fact one of the most widely used programming languages in scientific programming.

Fortran is considered to have passed through a "seven ages" evolutionary process [49] evolving simultaneously with distinct programming paradigms such as: Structured, Object Oriented, and Parallel Programming. It has been the first high level programming language with its own standard [6] which has evolved over the years:

- The revised and improved Fortran 66 [26] became Fortran 77 [4] in 1978.

- In 1992, a major revision of the standard, resulted in Fortran 90 [5].
- In 1997, a minor revision launched the Fortran 95 Standard [33].
- Likewise, Object-oriented features have been introduced in the Fortran 2003 Standard [34].
- Finally, the last Fortran Standard revision published as Fortran 2008 was released in 2011 [44], for which there are only a few fully compliant compilers.

A report on the compiler support for the newest Fortran standards (2003 and 2008) has been maintained up to date in the ACM SIGPLAN Fortran Forum, e.g. [14]. On the other hand, even when obsolete features were deleted by the standards committees in the different revisions, compatibility still remained: "Unlike Fortran 90, Fortran 95 was not a superset; it deleted a small number of so-called obsolescent features. However, the incompatibility is more theoretical than real as all existing Fortran 95 compilers include the deleted features as extensions" [42].

4 Specific Software Metrics and Methodology

Software metrics are in use since many years ago [16, 25, 48] for software analysis and evaluation, and project budget/resource estimation. We have selected some of the most useful ones and we have defined some specific ones for our purpose of current climate model software analysis.

4.1 Metrics

We have collected a number of source code measurements as a way of characterizing the current state of each program as well as the estimated effort involved in modernization, optimization, parallelization, etc. We have roughly classified source code measurements as follows:

- (S)LOC (Source Lines Of Code): LOC, Logical Executable LOC (LELOC), Non Blank Non Comments LOC (NbNcLOC), per (whole) model, file, module, and subprogram.
- File-related metrics: number of files, number of Fortran Functions and Subroutines per file.
- Subprogram Cyclomatic Complexity [45] as detailed in [10].
- Modules related measurements: number of Use statements (identifying those with and without "Only"), number of public and private subprograms (identifying functions and subroutines), number of module data declarations (*module variables*).
- Data related measurements: Common, Data and Block Data, and Equivalence statements, dummy arguments usage (number of arguments per subprogram, number of Intent In/Out/InOut arguments), number of local data and array declarations.

- Fortran Include and Preprocessor lines: number of Fortran Include lines, number of preprocessor lines (directives, conditional directives/constructs, macros, pragmas, errors), and specifically number of OpenMP directives.
- Loop related measurements: number of Do constructs, number of Do nested levels (1-10), number of *new style* Do loops (Do... End Do *format*), number of *old style* Do loops (labeled, no End Do).
- Go To and Deleted and Obsolescent Fortran Features: number shared Do loops, and number of Go To, arithmetic If, assigned Go To, computed Go To, Entry, Pause, and statement Function statements.
- Input Output operations: Number of Print, Read, Rewind, and Write statements.
- Other Fortran statements/features usage, such as the number of labels, Implicit, External, Format, and Call statements, If constructs and statements, etc.

Some of the above metrics are related to size and complexity of the software/climate models, e.g. lines of code (LOC, LELOC, etc.), number of files, cyclomatic complexity, and number of subprograms (Functions and Subroutines). All of these metrics help in having a reference or budget/resources estimation required for software modification, updating, optimization, etc. We can also relate some metrics to modularization and structured programming (or lack of), such as those about subprograms, modules, Go To and Fortran obsolescent and deleted features. Also, we focused rather specifically on metrics that we can use or relate to parallelization and HPC, such as those related to data declaration and processing (subprogram arguments, loops, global/static data in Fortran Common blocks).

4.2 Methodology

A priori, all of the metrics mentioned in the previuos section may seem too many to analyze each one of them in detail. However, we have found that the methodology we have used is conceptually and practically simple enough not only to have all of them at hand to be analyzed, but also to have other metrics as required by specific analysis/study. Our approach for collecting the metrics has been based on Abstract Syntax Trees (AST) compiler like source code analysis. In an AST, each node of the tree represents a programming construct occurring in the source code, and “the children of the node represent the meaningful components of the construct” [2].

We have extended Photran, an Eclipse plug-in specifically designed for Fortran [71]. Our extension has been focused on building a set of source code analyses performed by developing the visitor design pattern [27] (on Java) through the implementation of a set of (Java) classes to perform the required analysis on the AST structure. The specific steps followed to perform the measurements are listed below:

1. Source Code Obtaining: we have selected the climate models listed in the CMIP 5 web page [72]. We have downloaded the models with source code freely available on the internet. When the model source code was not

available for downloading, an email was sent to the owner group requesting their collaboration. This stage took more than a year, and as a result, we have collected the set of climate models shown in Table 1 above.

2. Metrics Gathering Process: The information regarding each file, module, and routine has been obtained by traversing the AST structure and collecting the above listed data about each Fortran measured item (see Figure 1). While the AST is constructed by Photran, several specific visitors have

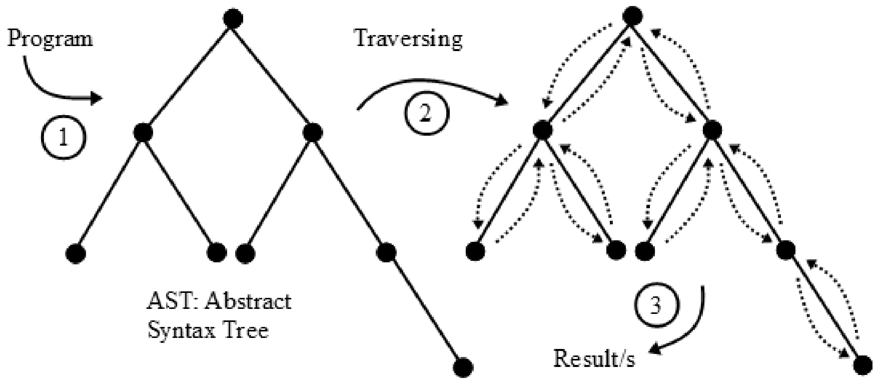

Figure 1: Measurement Process from Source Code To Metric Data

been implemented for computing metrics. Every visitor is independent of each other, and most of them are simple enough so that we were able to develop the complete set in less than two weeks.

3. Source Code Metrics Export/Storage: The output of the measurement process is stored in files. Some files are directly selected by the user to be examined from within the IDE (e.g. in CSV or HTML format). This enables the user to access data in a human readable format.

4. Source Code Metrics Import to DBMS: In order to obtain an efficient data analysis procedure and fast information recovery, an open source SQL DBMS has been used. Data stored in CSV files have been imported into the DBMS.

Even when the previous four steps have been carried out for a specific set of programs (climate models), involving source code in a single language (Fortran), and in a specific IDE (Photran), it is possible to follow the same steps in almost any other HPC application field.

5 Results

The methodology described in the previous section has allows to collect a vast amount of data. We will show only a set of well known metrics in order to describe the most interesting characteristics of HPC code and climate models in current production environments. Other information not included in this section is readily available upon request.

5.1 Size and Complexity Related Results

As a first step we have quantified the size of each model by counting the lines of code analysed: LOC, NbNcloc, and LELOC per model, and the results are shown in Table 2, which is ordered by LOC. In total, we have analysed about 7.3M LOC, composed by about 4.2M NbNcLOC, including about 2.6M LELOC. Also, Table 3 shows the models ordered by the number of subprograms, detailing

Model	LOC	LELOC	NbNcLOC
GISS	39950	14950	19631
CSIRO-Mk3.6.0	86494	35025	53340
INM-CM4	91337	47098	73946
GFDL-CM2.1	288153	94029	146218
CCSM3	361107	102584	186143
GEOS-5	367483	144598	211793
IPSL	374766	115554	180997
ModelE	379860	165866	278563
CMCC-CESM	380339	149811	217661
BCC-CSM1.1	451464	152236	235552
MPI-ESM-LR	477682	185426	283041
CFSv2-2011	478450	209139	297080
HadGEM2	633713	187547	343568
HadGEM3	737326	240533	439393
CCSM4	821726	262358	415608
CESM1	1370578	481871	803022
total	7340428	2588625	4185556

Table 2: Lines of Source Code per Climate Model: LOC, LELOC, NbNcLOC

functions (and how many of them are defined in module/s), and subroutines (and how many of them are defined in module/s). Thus, we have analyzed a total of about 51.2K Fortran subprograms, where approximately 42.5K are subroutines and 8.7K are functions. Even when data in Table 2 and Table 3 are shown here mostly for a raw idea of models size, they can be used for a more detailed and thorough analysis. Taking into account the different places for models HadGEM2 and HadGEM3 in both Tables (2 and 3 respectively), it is possible to conclude that they have larger subprograms than the other models, i.e. more LOC per subprogram, in average. Also, Table 3 shows a different usage of the very useful Fortran 90/95 capability: modules. Some models have extensively taken advantage of modules, such as GFDL-CM2.1, CCSM4, and CESM1, while others have only a little fraction of subprograms in modules or do not use modules at all, such as GISS, CSIRO-Mk3.6.0, INM-CM4, and HadGEM2.

In Table 4, the subprograms of each model have been distributed in a set of four classes according to their cyclomatic complexity (CC) ("the number of paths through a program" [45]), where the thresholds are defined as in [11]. Even though CC analysis/characterization has its detractors [62, 61], some researchers claim that the higher the level of CC of a procedure, the higher the probability of finding bugs or faults in it [15]. Moreover, each range of CC has been characterized with a corresponding routine risk [1] as shown in Table

Model	Subpr.	Func.	In Mod.	Subr.	In Mod.
GISS	143	18	0	125	0
CSIRO-Mk3.6.0	299	3	0	296	0
INM-CM4	739	42	0	697	45
GFDL-CM2.1	2012	331	329	1681	1670
HadGEM2	2032	89	14	1943	319
HadGEM3	2566	222	184	2344	1219
CCSM3	2740	327	320	2414	1950
CMCC-CESM	2822	524	451	2298	1360
GEOS-5	3171	721	487	2450	1645
IPSL	3361	441	413	2920	2222
MPI-ESM-LR	3410	453	393	2957	2198
CFSv2-2011	3774	1113	818	2661	1248
BCC-CSM1.1	3784	802	781	2982	2090
ModelE	3944	619	474	3325	1697
CCSM4	6424	1150	1117	5274	4649
CESM1	9832	1852	1809	7980	7516
total	51053	8707	7590	42534	29828

Table 3: Number of Subprograms per Model

5. Taking into account Table ccRangeT and Table 5, each model has a large number of subprograms with a high risk, and many subprograms considered as “untestable”. Even when we do not necessarilly accept the characterization given in Table 5 (which would lead to claim that every model has untestable code), we think that at least higher CC implies higher risk and, for example, stronger readability trouble.

5.2 Subprograms’s Metrics

Given the importance of per-subprogram work for maintenance in general and for optimization and parallelization in particular, we have collected several measurements of each models’ subprograms, some of them are shown in this subsection. Each model is ordered in Table 6 according to its maximum LOC subroutine (including the subroutine name) as well as the average subroutine’s LOC per model.

The number of parameters (Fortran arguments) a subprogram should have depends on many different factors. One of them is the programming language, whether it has been designed for managing complex data types or not. Also, the number of parameters is related to other classical sofware engineering characterization of software and subprograms in particular, such as the degree of coupling and/or cohesion of a program (intuitively, a greater the number of parameters corresponds to a greater degree of coupling and/or cohesion). Several standard tasks on HPC software in particular are also strongly related to the number of parameters: more parameters usually imply more data to process or related to processing, which also impacts optimization and parallelization tasks. Table 7 shows the models ordered by the maximum number of parameters (Fortran arguments) for a single subprogram. The average number of subprograms argu-

Model	0-10	11-20	21-50	> 51
GISS	62	26	34	21
CSIRO-Mk3.6.0	116	63	65	55
INM-CM4	459	135	93	52
GFDL-CM2.1	1442	249	212	109
HadGEM2	1003	361	383	285
HadGEM3	1318	421	453	274
CCSM3	2053	335	289	63
CMCC-CESM	1806	438	381	197
GEOS-5	2301	427	308	135
IPSL	2573	375	284	129
MPI-ESM-LR	2191	531	454	234
CFSv2-2011	2397	511	603	263
BCC-CSM1.1	2705	527	422	130
ModelE	3026	459	312	147
CCSM4	4682	803	701	238
CESM1	7312	1223	947	350

Table 4: Procedures with Cyclomatic Complexity in a Given Range (per Model)

Cyclomatic Complexity	Risk Evaluation
1-10	A simple module without much risk
11-20	A more complex module with moderate risk
21-50	A complex module of high risk
51 and greater	An untestable program of very high risk

Table 5: Cyclomatic Complexity Range Description

ments is also included in Table 7 for each model. While not invariant, models with higher maximum number of arguments for a single subprogram usually have higher average number of arguments.

Whereas there is no formal limit on the number of parameters that a procedure must have, the higher amount of parameters that a procedure has, the higher the understanding analysis that needs to be carried out. Different studies suggest that human beings cannot process a vast amount of information at the same time; moreover, studies claim that a human being cannot handle more that seven plus minus two "chunks of information" at the same time [51], or even a lower number [17]. Going in this direction there is a recommendation to maintain the number of subprograms parameters bound by 7 [46]. Table 8 shows the models ordered by the % of subprograms with 0-10 Fortran arguments. Taking into account the above, most of the subprograms are in a "safe" range of parameters number(7 ± 2), but there is an important number of subprograms which have their number of parameters between 11-20 and 21-50. Also, there is a small (but nonetheless important) set of subprograms exceeding 51 parameters.

Model	(Subroutine Name) max LOC	avg LOC
GISS	(SURFCE) 1333	156
CSIRO-Mk3.6.0	(TRACER) 1667	243
INM-CM4	(SEAICE_MODULE_SPLIT) 2018	128
CCSM3	(radcswmx) 2406	99
BCC-CSM1.1	(radcswmx) 2448	88
GEOS-5	(diaglist) 2602	81
ModelE	(init_ijts_diag1) 2975	83
CFSv2-2011	(INITPOST_GFS) 3267	108
CMCC-CESM	(DGESVD) 3409	123
MPI-ESM-LR	(datasub) 3504	123
IPSL	(physiq) 3848	75
HadGEM2	(GLUE_CONV) 4161	275
CCSM4	(hist_initFlds) 4623	99
GFDL-CM2.1	(morrison_gettelman_microp) 5124	108
HadGEM3	(conv_diag) 5602	261
CESM1	(lw_kgb03) 11975	115

Table 6: Models Maximum and Average LOC per Subprogram

5.3 Go To and Deleted and Obsolescent Fortran Features

One remarkable feature of the Fortran evolution lies in the fact that each new version is backward compatible with all the previous standard versions [49]. Furthermore, since the Fortran 90 standard revision, some features of the language have been marked as obsolescent and some of them have been deleted, but most compilers maintain them as valid (aka *Fortran extensions*). We have included Go To usage in some way *related to* deleted and obsolescent features (even when no one Fortran standard does so) because in several models Go To tends to produce spaghetti. While following the Fortran Standards Committees (i.e. the actual standards) Go To cannot be considered *obsolescent*, we do consider spaghetti code as *obsolescent*. There are opposing views about the Go To statement usage since long time ago: some pro-Go To-usage lead by [39] and the against-Go To-usage led by [19]. As always, we have to acknowledge that Go To usage does not generates spaghetti code by itself, but easily allows to produce such source code.

In Table 9 we have gathered the information concerning the number of occurrences and uses of deleted and obsolescent features (which appear in the appendix B of the Fortran standard [44]), in each models' source code. The first interesting result is the fact that all models make use of one of these features at least once. Another interesting result shown in Table 9 is that 10 out of 16 models make use of the Arithmetic If statement. Finally, two deleted features appearing in the appendix B of the Fortran 2008 standard and in the Fortran 90 standard, namely Pause and Assigned Go To statements, are still found in the source code of 6 models. Furthermore, we have found two models containing both deleted features. Models in Table 9 are ordered by the total number of obsolescent and deleted Fortran features, and Table 10 shows the models ordered by the number of Go To statements. A further and more complicated analysis about Go To, and deleted and obsolescent Fortran features remains, basically

Model	Subprogram	Max. Args. #	Avg. Args. #
GISS	KPPMIX and MSTCN2	21	3
INM-CM4	TRAN_ICE_MPDATA	35	4
ModelE	Lagrangian_to_Eulerian	52	4
CSIRO-Mk3.6.0	c_aero1	63	7
CMCC-CESM	IGSCINT	64	5
CCSM3	slttraj	68	4
GFDL-CM2.1	moist_processes	80	6
IPSL	cv3a_compress	95	5
CCSM4	seq_infodata_PutData	104	4
CESM1	seq_infodata_GetData	117	4
CFSv2-2011	RCtoNamSfcNamelist	117	6
GEOS-5	CATCHMENT	163	5
MPI-ESM-LR	update_surface	166	5
BCC-CSM1.1	CAM_V5	183	5
HadGEM2	Eot_diag	333	19
HadGEM3	Eot_diag	378	16

Table 7: Maximum and Average Number of Arguments per Procedure per Model

related to how *dangerous* are those statements on the source code. Even when *dangerousness* is hard to measure (such as the so called *bad smell* is), having a deleted or obsolescent feature in a routine which is not used in most of the program executions is not the same as having those statements in a routine used 50% of every run.

5.4 Common Blocks and Equivalence

Fortran common blocks are "blocks of physical storage ... that may be accessed by any of the scoping units in an executable program" [5]. This language feature has been defined and used since Fortran 77 due to the fact that Fortran 77 does not have global variables, among other reasons. There are some factors that discourage the usage of common blocks:

- For a common block to be available, it must be declared every time it is used in every routine that makes use of it. This drives people to use common blocks via an INCLUDE statement or a #include C preprocessor line. Thus, the common block is replicated on in each subroutine where it is used, even when not every variable defined in the common block is actually needed in the subroutine.

- In a common block, the variables do not need to have the same names in each place in which they occur, as shown in the example below:

Model	0-10 (%)	11-20	21-50	51 -100	101 -200	> 200
HadGEM2	1132 (56)	406	320	109	60	5
HadGEM3	1648 (64)	366	367	124	52	9
CSIRO-Mk3.6.0	234 (78)	35	27	3	0	0
GFDL-CM2.1	1717 (85)	217	70	8	0	0
CMCC-CESM	2431 (86)	289	96	6	0	0
MPI-ESM-LR	2959 (87)	309	131	9	2	0
CFSv2-2011	3340 (89)	340	81	10	3	0
GEOS-5	2807 (89)	277	86	0	1	0
INM-CM4	663 (90)	67	9	0	0	0
IPSL	3021 (90)	224	99	17	0	0
BCC-CSM1.1	3410 (90)	231	127	12	4	0
CCSM4	5962 (92)	325	129	6	2	0
CCSM3	2527 (92)	136	74	3	0	0
CESM1	9145 (93)	479	183	22	3	0
ModelE	3693 (94)	204	46	1	0	0
GISS	139 (97)	2	2	0	0	0

Table 8: Number of Subprograms per Model in a Given Range of Fortran Arguments Number

```
Subroutine Sub1                Subroutine Sub2
  Common /C1/ A, I, B            Common  /C1/ C, J, G

  A = A + B                      C = C + G
  I = I + A                      J = J + C

  Print *, A, I, B               Print *, C, J, G
End                            End
```

and variable aliasing problems are implicit as well as those corresponding to data representation.

- Combined with implicit data declaration and the usage of the EQUIVALENCE statement, the common blocks could be very complex to analyse and understand.

Table 11 shows Fortran common blocks and the number of Equivalence statements found in each model ordered by the number of common blocks declared per model. Most of the models declare common blocks in files later included in several subroutines/functions, so it is possible that a single block is used several times, including cases where not all of the variables declared in the common block are actually used.

5.5 Loops

There is a general rule which indicates that 80% of the time is spent in 20% of the source code, and even referred to as the *90/10 Locality Rule* [30]. Taking this into consideration, restructuring or improving 20% of the source code could

Model	Arith. If	Assig. Go To	Comp. Go To	Entry	Pause	Total
HadGEM3	0	0	0	0	1	1
CCSM3	0	0	1	0	0	1
GFDL-CM2.1	1	0	1	0	0	2
HadGEM2	0	0	0	0	2	2
IPSL	0	0	0	0	2	2
BCC-CSM1.1	0	0	3	0	0	3
CCSM4	1	0	10	8	1	20
CMCC-CESM	4	0	11	5	1	21
MPI-ESM-LR	3	4	9	4	1	21
CSIRO-Mk3.6.0	13	0	0	16	0	29
CESM1	16	6	9	8	1	40
INM-CM4	35	1	4	0	0	40
GEOS-5	37	0	22	13	0	72
GISS	33	0	6	36	0	75
CFSv2-2011	31	0	24	94	0	149
ModelE	23	0	4	139	0	166

Table 9: Fortran Obsolete Features Found within Each Model

provide a speed up that justifies applying changes to the source code. Consequently, a key problem is how to identify those areas to be improved. Even though Do loop statements are the best candidates to apply optimizations to, they must be located and analysed first, and then transformed. Fortran 90 introduced the End Do statement in order to structure the finalization of the Do loop statement. The End Do statement has been available to be used since Fortran 90, but we have found very frequently the usage of the old style Do loop statements -that ending in a label- as shown in Table 12. Furthermore, we have found shared Do loop termination, which has been identified as an obsolete Fortran language feature since the Fortran 90 Standard [5]. Models in Table 12 are ordered by the proportion of New/Old Do loops usage, and we have included the number of shared Do loop termination in order to have an idea of potential problems. Even in those models with a *good* proportion of new style Do loops (e.g. more than 80%), there are old style Do loops with shared Do loop termination.

5.6 User Defined Types

Large programs have a high level of complexity, which becomes a problem taking into account level of complexity that any person can deal with at any one time: “The amount of complexity that the human mind is able to cope with at any instant in time is considerably less than that embodied in much of the software one might build”, [18]. Data abstraction is proposed and used as a mechanism to reduce the levels of complexity, allowing humans to focus on the key problem [28]. Thus, abstraction helps to reduce software complexity and as a consequence it improves the process of software understanding. In other words, “An abstraction is a simplified description of a system that emphasizes the system’s important characteristics and ignores those details immaterial to an understanding of the system at a given level” [32]. This concept has been

Model	Go To
IPSL	150
CCSM3	176
GFDL-CM2.1	183
CMCC-CESM	189
CSIRO-Mk3.6.0	240
HadGEM3	264
GEOS-5	325
BCC-CSM1.1	370
HadGEM2	474
INM-CM4	490
CCSM4	589
CESM1	806
MPI-ESM-LR	897
ModelE	1170
GISS	1173
CFSv2-2011	3078

Table 10: Number of Go To Statements per Model

applied by software engineers at least through the last three decades. We have counted the user-defined data types found in the models as one of the possible measures of the abstraction concept. Consequently, we have counted each occurrence of the "Type" statement on each model.

In Table 13 the amount of (Fortran derived) type definitions per model is shown. Surprisingly, two models do not have any user-defined data type, which indicates that they perform the entire simulation using only Fortran native data types. Even when the *raw* numbers in Table 13 are rather impressive, a further analysis can be made taking into account NbNcLOC. In the case of CSIRO-Mk3.6.0, for example, more than 19000 NbNcLOC are involved without a single user defines type. As another example, the model CESM1 only 458 user defined types are involved in more than 803000 NbNcLOC. In general, the number of LOC of each model indicate that there should be much more user defined types than those actually used.

5.7 Argument Intent

Fortran routine Argument Intent has been available in Fortran since 1992. Claiming that the use of intent specification attribute purpose is just adding documentation to the intended use of the dummy arguments would be an understatement. This point was covered in Note 5.14 from the Fortran 2003 standard [5]:

> ... "Argument intent specifications serve several purposes in addition to documenting the intended use of dummy arguments. A processor can check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argument could possibly be referenced before it is defined. If the

Model	Common	Equivalence
CCSM3	0	4
GFDL-CM2.1	0	1
CMCC-CESM	0	0
HadGEM3	1	0
IPSL	1	0
GEOS-5	10	30
HadGEM2	44	5
MPI-ESM-LR	44	1
ModelE	84	15
CCSM4	91	4
CESM1	93	14
BCC-CSM1.1	147	36
GISS	155	2
CFSv2-2011	221	184
CSIRO-Mk3.6.0	330	51
INM-CM4	1114	5

Table 11: Common Block and Equivalence Usage

> procedur's interface is explicit, the processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT (INOUT) dummy arguments are definable. A more sophisticated processor could use this information to optimize the translation of the referencing scoping unit by taking advantage of the fact that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and that any prior value of an actual argument corresponding to an INTENT (OUT) dummy argument will not be referenced and could thus be discarded"...

Table 14 shows that the usage of the intent specification attribute is not utilized to its full potential. Evidence shows that the many models which use the intent attribute specification do not make an intensive use of such language attribute (e.g. in less than 50% of the arguments), and other models do not make any use of these features at all. Rather surprinsingly, models with a large number of subprogram arguments such as HadGEM2 have used attributes for less than 50% of them. Summarizing the data in Table 14, models have used attributes on arguments in a range from 0% to (GISS and INM-CM4) to 97% (GFDL-CM2.1) of the routines arguments, and only 6 of 16 have used attributes on more than 70% of the arguments.

5.8 Preprocessing Directives

The C preprocessor is the macro preprocessor for the C language [37], also known as *cpp*. It was initially developed to be used by the C compiler. Nowadays its use has been broadly extended to other programming languages, including Fortran. The C preprocessor's objective is to be a "textual preprocessor" [23]. Preprocessing is carried out before the compilation stage by replacing some chunks of text with others in the program source code. The C preprocessor has

Model	Num. Do	% New	% Old	Shared
HadGEM3	30240	100	0	0
CCSM3	7613	99	1	2
HadGEM2	23251	98	2	60
GFDL-CM2.1	6367	98	2	6
CESM1	23316	97	3	14
CCSM4	16981	97	3	16
BCC-CSM1.1	11289	95	5	125
GEOS-5	7287	86	14	127
ModelE	13371	82	18	956
IPSL	11505	80	19	206
MPI-ESM-LR	11502	78	21	75
INM-CM4	5269	72	23	278
CFSv2-2011	14556	75	24	312
CMCC-CESM	9614	72	27	69
CSIRO-Mk3.6.0	4583	40	60	1375
GISS	1325	2	98	388

Table 12: Do Statement Usage

been used by programmers to integrate configuration management and program portability [64, 23]. We believe that there are two important aspects to study the preprocessor directives usage in Fortran. The first aspect is the fact that using the preprocessor increases the program's complexity and hence the work needed to understand it [22, 3, 31]. The second important aspect is the underlying complexity of restructuring programs in the presence of preprocessor directives, which still remains a challenge.

Table 15 shows the number of preprocessing directives per model. We also analysed the number of Fortran Include statements used, wich are shown in the column labeled "Fortran Inc." in Table 15. Fortran Include statements are not C preprocessor directives, but we consider them as preprocessing directives since the semantic is the same as that of a #include C preprocessor directive. Some of the results shown are rather expected, such as having a low number of preprocessing directives in the models with less LOC, e.g. GISS, INM-CM4, and CSIRO-Mk3.6.0. Also, only four models have a relatively high Fortran Include statements: INM-CM4, CSIRO-Mk3.6.0, IPSL, and HadGEM2 have a number of Fortran Include statements which is at least a quarter of the total number of preprocessing directives. Conditional directives (#ifdef, #ifndef, #if, #else, #elif, #endif) constitute the largest fraction of the total number of preprocessing directives in almost all models. Other preprocessor directives such as #error and #line are not found in any model.

The uses of the preprocessor directives could be considered at least non standard, though widely used. In some cases we found extremely *non orthodox* usage, such as

```
subroutine1 ( parameter1,parameter2
 #include additionalParameters
 )
```

Model	Type Def.
GISS	0
CSIRO-Mk3.6.0	0
HadGEM2	26
CMCC-CESM	46
HadGEM3	62
CFSv2-2011	97
IPSL	100
ModelE	106
GFDL-CM2.1	129
GEOS-5	164
MPI-ESM-LR	165
CCSM3	169
BCC-CSM1.1	228
CCSM4	341
CESM1	458

Table 13: Type statement usage

Model	Arguments	Intent In	Intent Out	Intent InOut
GISS	380	0	0	0
INM-CM4	1748	0	0	0
CSIRO-Mk3.6.0	2209	19	7	0
IPSL	8988	1679	677	287
HadGEM3	9829	4353	718	1045
CMCC-CESM	10068	3984	611	654
GFDL-CM2.1	11390	7756	1923	1339
CCSM3	11708	6971	1825	845
GEOS-5	12712	6306	2617	1156
ModelE	14508	6640	1263	1186
MPI-ESM-LR	18167	7131	1600	1186
BCC-CSM1.1	19650	10747	2405	1361
CFSv2-2011	22661	8111	1676	860
CCSM4	28230	16499	4269	3038
HadGEM2	38353	8368	1614	1953
CESM1	41744	24663	6653	5200

Table 14: Arguments Intent Specification Usage

Model	Total	Conditional	Fortran Inc.	Prep. Include
GISS	0	0	0	0
INM-CM4	18	2	16	0
CSIRO-Mk3.6.0	37	6	31	0
GEOS-5	1795	1446	184	165
CFSv2-2011	1827	1243	65	519
GFDL-CM2.1	2056	1834	138	84
ModelE	3316	3202	75	39
CMCC-CESM	3616	3350	139	127
IPSL	5121	3346	1315	460
MPI-ESM-LR	5550	5223	145	132
CCSM3	5671	4166	207	1298
HadGEM3	6390	4575	927	888
BCC-CSM1.1	8375	6093	474	1808
CCSM4	10283	9466	465	352
HadGEM2	12893	5852	3501	3540
CESM1	13300	12563	430	307

Table 15: Preprocessing Directives per Model

which can be considered as highly error prone and unreadable. Also, new and current Fortran standard features can be used, such as argument association by argument keyword and optional dummy arguments.

6 Discussion

Some data gathered in this research seems to validate interesting results obtained in previous works about scientific software and also show some new and specific results on GCM. The first one seems to be the slow pace at which Fortran source code and Fortran features usage evolve. While some language features have been marked as obsolete in the Fortran standard more than 20 years ago [5] they are still being used. This may reinforce the idea that programming skills and knowledge are passed on from scientist to scientist [70, 8]. One aspect to emphasize is the usage of Go To statement on every model, a thorough study should be performed in order to characterize this Go To statement usage. Another reason to avoid modernizing Fortran code is based on "If it ain't broke, don't fix it" rule. However, even if not broken, unreadable and hard to change (among other characteristics of legacy) software is rarely ready to take advantage of lots of new facilities and computing power of new hardware. Furthermore, the highly parallel new hardware (e.g. multi-many core processors) requires a specific bias in software development and maintenance towards parallel computing [63].

A second interesting point is highlighted by the values obtained for some programming characteristics like the number of routines' parameters (Fortran arguments). Each model has at least one routine in the range of 21-50 parameters, that could be considered as a very high value as regards the number of parameters that a routine should have. Furthermore, we have found that the model which has the lowest maximum amount of parameters in a routine is

21 but the high maximum is 378 (see Table 7). The range is huge, and the corresponding restructuring work would be correspondingly huge too.

Another interesting value found was the number of lines per routine, for which the minimum average per model is 75 lines per routine and a maximum average of 275 lines per routine. The routine with the absolute highest value in lines of code has 11975 lines (see Table 6). It might be indicating that some modularization techniques should be implemented by computing scientists [8, 58, 12]. A third remarkable aspect that could be extracted taking into account Table 4 and Table 13 is the possible cyclomatic complexity reduction that could be made by using more data types, thus taking advantage of data abstraction for decreasing some complexity aspects. A fourth observed feature in the source code is the intensive use of the C preprocessor directives and Fortran include directive (see Table 15) for including common block definition, for portability, for managing configurations activities, and so forth.

Finally, the data collected seems to validate the existence of a "mismatch" or a *gap* between scientific software production and guidelines and good practices suggested by the software engineering community [35, 8]. These results enable us to determine the need of (semi)automated tools to transform certain not desired software features found in the source code in order to help it evolve. These transformations should allow developers to automatically eliminate obsolete features by the corresponding replacement, to introduce new language features to the source code, etc. More elaborate transformations could include the automatic inclusion of new user-defined data types.

7 Related Works

In 1971, Donal Knuth [38] analysed a set of Fotran programs "in an attempt to discover quantitatively what programmers really do". To perform such analysis he studied a variety of programs written by different people: a set of 400 programs distributed in 250000 punched cards. Knuth provided a thorough study of the Fortran 66 constructs' usage on those 400 programs. One remarkable aspect of the study was the handmade analysis that was performed. Further research attempted to analyse a set of 255 Fortran Programs by using a tool called FORTRANAL which performs the static analysis to gather information about 31 metrics [40]. The aim of this work was to find some correlation among different groups of metrics. The study carried out in [60] focused on finding Fortran programs' characteristics that were relevant in the parallelization process from the compiler writers view point, in order to contribute to data dependence analysis and program transformation. A series of articles studying the relationship between Fortran programs and complexity metrics were published by Victor R. Basili. In [9] complexity metrics were used to evaluate software quality, to estimate software cost, and to evaluate the stability and quality of a software product during the development process. In [10] a set of metrics were studied in order to determine the relationship among some software aspects such as effort, program bugs, among others. Software was analysed by using an automated tool called SAP.

Software metrics were also used to evaluate system maintainability. In [16] five methods for quantifying the maintainability of a program were studied. That work indicates that automated analysis can be used to "evaluate and compare software". Nowadays, software engineering techniques and principles are widely adopted in the software industry. However, as contradictory as it may sound, evidence seems to indicate that some of the problems that had already been overcome years ago in the "spaghetti code wars of the 1970s" strike back rather periodically [50].

Scientific software has been the subject of thorough studies [8, 58, 12, 29, 20, 35, 36, 70, 57]. All of those articles shed light on the fact that there is a mismatch, to say the least, between well-known and broadly accepted software engineering methods and practices and those methods and practices used on scientific software building processes.

8 Conclusions and Further Work

In this article we have studied clasical scientific software (a set of GCM programs), in order to analyse the way Fortran is used in the scientific software environment. Our approach has been based on traversing the AST program structure in order to obtain the required information. We have collected data regarding different program aspects such as routine complexity, number of parameters, routine length, abstraction usage, obsolete language features, and C preprocessor usage among others.

Our approach is based on gathering the information in the precise moment the user requires it, no previous preprocessement is required. In addition, our measurement process has been integrated to a modern IDE, which implies that programming and measurement tasks can be performed at the same time and in the same environment.

The results indicate that software techniques adopted by the industry are different from those used in scientific software production. There is a strong need to adapt scientific software to new standards mostly to take advantage of new hardware processing facilities. The path is not straightforward, though: source code optimization and parallelization can be made with less effort if the source code is first made readable and, at least, less prone to error (e.g. by being -some Fortran- standard compliant).

We will continue our work following several guidelines, taking into account the work in this article as well as previous work on Fortran source code transformations and Fortran legacy software [47, 66, 67]:

- Implement new metrics to be applied in Fortran Source code.
- Integrate these metrics on modern IDE.
- Measure more Fortran source code in order to determine and prioritize which transformations should be applied on the source.
- Collect more data in order to determine new trends in scientific software production.

Also, source code metrics help in the analysis of how successful are source code transformations, since the original and transformed source code can be measured and, hence, objectively compared.

Acknowledgements

We would like to thank Beijing Climate Center, China Meteorological Administration, Commonwealth Scientific and Industrial Research Organisation in collaboration with the Queensland Climate Change Centre of Excellence, Institute for Numerical Mathematics, Institut Pierre-Simon Laplace, Met Office Hadley Centre, Max Planck Institute for Meteorology (MPI-M), NASA Goddard Institute for Space Studies, NASA Global Modeling and Assimilation Office, National Center for Atmospheric Research, Geophysical Fluid Dynamics Laboratory, National Science Foundation, Department of Energy, National Center for Atmospheric Research, Centro Euro-Mediterraneo per I Cambiamenti Climatici, Center for Ocean-Land-Atmosphere Studies and National Centers for Environmental Prediction for providing us with the information and model source code to achieve the research aims.

References

[1]

[2] Alfred V Aho et al. *Compilers: principles, techniques, & tools.* Pearson Education India, 2007.

[3] Zubair Akhtar and Wenbin Ji. On the relation of preprocessor directives & bugs.

[4] American National Standards Institute. X3. 9-1978. *American National Standards Institute, New York*, 1978.

[5] American National Standards Institute. *American National Standard for programming language, FORTRAN — extended: ANSI X3.198-1992: ISO/ IEC 1539: 1991 (E).* American National Standards Institute, September 1992.

[6] J. Backus. The History of Fortran I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, 1978.

[7] David W Balmer and Ray J Paul. Casm-the right environment for simulation. *Journal of the Operational Research Society*, pages 443–452, 1986.

[8] Victor R Basili, Daniela Cruzes, Jeffrey C Carver, Lorin M Hochstein, Jeffrey K Hollingsworth, Marvin V Zelkowitz, and Forrest Shull. Understanding the high-performance-computing community. 2008.

[9] Victor R Basili and Tsai-Yun Phillips. Evaluating and comparing software metrics in the software engineering laboratory. *ACM SIGMETRICS Performance Evaluation Review*, 10(1):95–106, 1981.

[10] Victor R. Basili, Richard W Selby Jr, and T Phillips. Metric analysis and data validation across fortran projects. *Software Engineering, IEEE Transactions on*, (6):652–663, 1983.

[11] Michael Bray, Kimberly Brune, David A Fisher, John Foreman, and Mark Gerken. C4 software technology reference guide-a prototype. Technical report, DTIC Document, 1997.

[12] Jeffrey C Carver, L Hochstein, Richard P Kendall, Taiga Nakamura, Marvin V Zelkowitz, Victor R Basili, and Douglass E Post. Observations about software development for high end computing. *CTWatch Quarterly*, 2(4A):33–37, 2006.

[13] Jeffrey C Carver, Richard P Kendall, Susan E Squires, and Douglass E Post. Software development environments for scientific and engineering software: A series of case studies. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 550–559. IEEE, 2007.

[14] Ian D. Chivers and Jane Sleightholme. Compiler support for the fortran 2003 and 2008 standards revision 14. In *ACM SIGPLAN Fortran Forum*, volume 32, pages 21–34, 2013.

[15] Istehad Chowdhury and Mohammad Zulkernine. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1963–1969. ACM, 2010.

[16] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[17] Nelson Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and brain sciences*, 24(1):87–114, 2001.

[18] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *Notes on structured programming.* Technological University Eindhoven Netherlands, 1970.

[19] E.W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[20] Steve M Easterbrook and Timothy C Johns. Engineering the software for understanding climate change. *Computing in Science & Engineering*, 11(6):65–74, 2009.

[21] Paul N Edwards. A brief history of atmospheric general circulation modeling. *International Geophysics*, 70:67–90, 2001.

[22] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *Software Engineering, IEEE Transactions on*, 28(12):1146–1170, 2002.

[23] J-M Favre. Preprocessors from an abstract point of view. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 329–338. IEEE, 1996.

[24] Louis Fein. The role of the university in computers, data processing, and related fields. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 119–126. ACM, 1959.

[25] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2):149–157, 1999.

[26] A. FORTRAN. X3. 9-1966. *American National Standards Institute Incorporated, New York*, 1966.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley Reading, MA, 1995.

[28] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[29] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8. IEEE Computer Society, 2009.

[30] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* 5th edition, 2011.

[31] Gerard J Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.

[32] John F Isner. A fortran programming methodology based on data abstraction. *Communications of the ACM*, 25(10):686–697, 1982.

[33] ISO. ANSI/ISO/IEC 1539-1:1997: Information technology — programming languages — Fortran — part 1: Base language.

[34] ISO. ANSI/ISO/IEC 1539-1:2004(E): Information technology — Programming languages — Fortran Part 1: Base Language. pages xiv + 569, May 2004.

[35] Diane F Kelly. A software chasm: Software engineering and scientific computing. *Software, IEEE*, 24(6):120–119, 2007.

[36] Richard Kendall, Jeffrey C Carver, David Fisher, Dale Henderson, Andrew Mark, Douglass Post, Clifford E Rhoades, and Susan Squires. Development of a weather forecasting code: A case study. *Software, IEEE*, 25(4):59–65, 2008.

[37] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[38] D.E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[39] Donald E Knuth. Structured programming with go to statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[40] Hon Fung Li and William Kwok Cheung. An empirical study of software metrics. *Software Engineering, IEEE Transactions on*, (6):697–708, 1987.

[41] Peter Lynch. The origins of computer weather prediction and climate modeling. *Journal of Computational Physics*, 227(7):3431–3444, 2008.

[42] Cohen Malcom. Fortran: A Few Historical Details. http://www.nag.co.uk/nag ware/NP/doc/fhistory.asp, October 2004.

[43] Syukuro Manabe and Kirk Bryan. Climate calculations with a combined ocean-atmosphere model. *J. Atmos. Sci*, 26(4):786–789, 1969.

[44] A. Markus. Design patterns and Fortran 2003. In *ACM SIGPLAN Fortran Forum*, volume 27, pages 2–15. ACM, 2008.

[45] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[46] Steve McConnell. *Code complete*. O'Reilly Media, Inc., 2004.

[47] M. Méndez, J. Overbey, and F.G. Tinetti. Legacy fortran software: Applying syntactic metrics to global climate models. In *XVIII Congreso Argentino de Ciencias de la Computación 2012*, pages 847–856, 2012. Available at https://lidi.info.unlp.edu.ar/˜fernando/FortranLegacy/.

[48] Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 83–86. ACM, 2001.

[49] M. Metcalf. The seven ages of fortran. *Journal of Computer Science and Technology*, 11(1):1–8, 2011.

[50] Tommi Mikkonen and Antero Taivalsaari. Web applications: Spaghetti code for the 21st century. 2007.

[51] George A Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[52] Ricardo Miranda, F Braunschweig, P Leitao, R Neves, F Martins, and A Santos. Mohid 2000, a coastal integrated object oriented model. *Southampton, UK: WIT Press, Hydraulic Engineering Software VIII*, 2000.

[53] Peter Müller. Constructing climate knowledge with computer models. *Wiley Interdisciplinary Reviews: Climate Change*, 1(4):565–580, 2010.

[54] J Pipitone and S Easterbrook. Assessing climate model software quality: a defect density analysis of three models. *Geoscientific Model Development Discussions*, 5(1):347–382, 2012.

[55] Miriam Schmidberger and Bernd Brugge. Need of software engineering methods for high performance computing applications. In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pages 40–46. IEEE, 2012.

[56] Judith Segal. Models of scientific software development. *SECSE 08, First International Workshop on Software Engineering in Computational Science and Engineering, Leipzig, Germany*, May 2008.

[57] Judith Segal. Some challenges facing software engineers developing software for scientists. In *2nd International Software Engineering for Computational Scientists and Engineers Workshop (SECSE '09), ICSE 2009 Workshop, Vancouver, Canada*, pages 9–14, May 2009.

[58] Judith Segal. Scientists and software engineers: A tale of two cultures. *Proceedings of the Psychology of Programming Interest Group, PPIG 08*, pages 10–12, September 2008.

[59] M Shaw et al. Computer science: Reflections on the field, reflections from the field, 2004.

[60] Zhiyu Shen, Zhiyuan Li, and P-C Yew. An empirical study of fortran programs for parallelizing compilers. *Parallel and Distributed Systems, IEEE Transactions on*, 1(3):356–364, 1990.

[61] M Shepperd and Darrel C Ince. A critique of three metrics. *Journal of Systems and Software*, 26(3):197–210, 1994.

[62] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[63] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm, (update 2009).

[64] Andrew Sutton and Jonathan I Maletic. How we manage portability and configuration with the c preprocessor. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 275–284. IEEE, 2007.

[65] K.E. Taylor, R.J. Stouffer, and G.A. Meehl. An overview of cmip5 and the experiment design. *Bulletin of the American Meteorological Society*, 93(4):485, 2012.

[66] F. G. Tinetti and M. Méndez. Fortran legacy software: source code update and possible parallelisation issues. *ACM SIGPLAN Fortran Forum*, 31(1):5–22, April 2012.

[67] Fernando G. Tinetti, Mariano Méndez, and Armando de Giusti. Restructuring fortran legacy applications for parallel computing in multiprocessors. *The Journal of Supercomputing*, 64(2):638–659, 2013.

[68] Warren M Washington, Lawrence Buja, and Anthony Craig. The computational future for climate and earth system models: on the path to petaflop and beyond. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1890):833–846, 2009.

[69] Spencer Weart. The development of general circulation models of climate. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, 41(3):208–217, 2010.

[70] Gregory V Wilson. Where's the real bottleneck in scientific computing? *American Scientist*, 94(1):5, 2006.

[71] Photran, an Integrated Development Environment and Refactoring Tool for Fortran. http://www.eclipse.org/photran/.

[72] Coupled Model Intercomparison Project Phase 5. http://cmip-pcmdi.llnl.gov/cmip5/.

[73] Some Work on Fortran Legacy Code. https://lidi.info.unlp.edu.ar/~fernando/FortranLegacy/.