5th International Symposium on Three-Dimensional Electromagnetics
Sapporo, Japan, May 7-9, 2013

# Programming in 3-D electromagnetics: look from outside

Alexey Geraskin
Earthquake Research Institute, University of Tokyo

## SUMMARY

For the last few decades incredible progress in the area of computer science has been achieved both in hardware and software developments. Software development by itself has become a standalone and very complex science. Programming technologies went far away from the early times and continue to progress very rapidly. Nowadays these technologies make it possible to develop software much more efficiently and in general without pain and frustration. On the other hand, a progress in EM forward and inverse modelling of the 3-D problems with realistic levels of complexity and spatial detail is still not satisfactory. Still computational loads to tackle realistic scenarios appear to be prohibitively demanding due to memory and speed limitations. And very often such limitations are caused not only by hardware deficiency, or/and inefficiency of the used numerical algorithms, but also by immaturity of the codes which implement these algorithms. Author's analysis of some 3-D EM codes led him to conclusion that most of them are substantially out-of-date and written in a rather inefficient and naive way. This causes not only the serious performance issues, but also makes existing codes very difficult to understand, maintain, and introduce new features. In this abstract some ideas about modern efficient coding are discussed.

**Keywords**: computer science, effective programming, parallel computations, optimization, refactoring

## INTRODUCTION

Electromagnetic (EM) studies of the conducting Earth, from the near surface to regional and global, have advanced significantly over the past few years. This progress has been achieved by the increased accuracy, coverage and variety of the newly available data sets, as well as by the new developments in the methods of three-dimensional (3-D) modeling and 3-D inversion of EM data. However still computational loads to tackle 3-D problems with realistic levels of complexity and spatial detail on a routinely basis appear to be prohibitively time-consuming due to memory and speed limitations. The mainstream solutions to improve the situation include: a) the use of more powerful computers; b) parallelization of the codes; c) development/implementation faster numerical algorithms. The first and second options are often seems rather trivial, and most of literature concentrates on elaboration and discussion of the numerical approaches which help to speed up the solutions. Unfortunately small attention is paid in community to effective coding of the proposed numerical concepts. From author's point of view the effective coding often delivers more optimality to the resulting codes than the implementation of the advanced numerical algorithms. Moreover such coding helps to minimize the efforts and save the nerves while testing the codes. In this abstract some basic ideas about modern efficient coding are presented.

## CHOOSING APPROPRIATE LANGUAGE

As far as author knows most of 3-D EM codes are written in Fortran. In general modern Fortran is rather convenient for mathematical calculations. The reasons for that are as follow: (1) Fortran has vector operations syntax; (2) Fortran does not use pointer arithmetic; this makes memory management easier and, the most important, gives possibility to Fortran compiler to make very effective optimization. As for Fortran compilers, nowadays most of them have a possibility for auto-parallelism. But this feature is strongly compiler-dependent and requires very accurate coding. Intel Fortran Compiler can produce (partly because of (2)) the most efficient code for the latest processor architectures; other Fortran compilers are not so efficient. The latter means that it is very important to have the latest version of the compiler.

But, following Pareto principle (Joseph, Frank, 1988), only small amount of code perform time-consuming calculations. All other parts of the code do some supporting work: read input data, write results, manipulates with the data. This type of work usually does not take significant time during the code run, but can take much time and nerves during the coding. Fortran is not ideal language for such type of work. The point is that changing language to a more suitable one (say to, Java (not to be confused with JaveScript), C#, C++) may lead to more accurate, clean and understandable code. And ultimately the process of the

code development will be not painful (as often happens), but indeed enjoyable.

## OBJECT-ORIENTED PROGRAMMING

Object oriented programming (OOP) is a programming paradigm which operates with *objects* as a main concept. Objects possess data fields which describe its current state. Also objects have associated procedures which are used for changing its state. Objects interact with each other in some way, which is defined by program architecture. Objects as a formal concept in programming were introduced in the 1960s in Simula 67 (Holmevik, 1994). The Smalltalk language, which was developed at Xerox PARC (Goldberg, Kay, 1976) in the 1970s introduced, the term object-oriented programming to represent the pervasive use of objects and messages as the basis for computation. In the 1990s OOP becomes dominant programming methodology. Programming languages supporting this technique became widely available. Now a great experience of using OOP paradigm has been accumulated. Part of this experience led to elaboration of design patterns (DP) concept (Gamma, et al., 1995). In general, DP can be used to refer to any general, repeatable solution to a commonly occurring problem in software design. But especially in OOP design patterns became very influential. However, it is very important to understand that OOP is just the instrument, which nowadays becomes very complex. Using it without appropriate understanding and experience could lead to insufficient solutions.

Fortran has started to support OOP only a few years ago (Worth, 2008). And if truth to be told, still this support in is far from optimum. Moreover in case of intensive computation OOP paradigm seems to be superfluous. But, as mentioned before, only small part of code does intensive calculations. So, again, it seems that using other language for non-calculation part of the code is more appropriate.

## USING IDE

Integrated Development Environment (IDE) – is integral part of software development in a pair with a compiler. Some of IDEs are: Eclipse, IntelliJ IDEA, MonoDevelop, Microsoft Visual Studio, etc. Main parts of IDE usually are: source code editor, build automation and a debugger. Modern source editors have many useful functions, such as syntax highlighting, code auto completion, snippets (small region of re-usable code) support. Using IDE makes development much easier and faster.

## OPTIMIZATION

Often it is very difficult without special tools to find an exact place in the code where performance bottleneck appears. In the most cases it is located in other place than developer assumes. To solve this problem profilers (e.g. Intel VTune Performance Analyzer) should be used. Profiler works simultaneously with examined program and calculates different performance metrics. These metrics could be from simpler ones like time consuming, memory consuming, cores utilization (on multicore processors) up to very architecture-specified metrics. After such analysis, possible bottlenecks could be eliminated in the most efficient way. But still the process of elimination is very creative by itself.

## CODE REFACTORING

Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior" (Fowler, 1999). Refactoring reduces complexity, improves maintainability and readability of the source code. Moreover refactoring should be done not once, but periodically. In this case it will take small amount of time each time, but will lead to significant improvement of the code. Unfortunately, many codes in the field of 3-D electromagnetics have been never refactored being very tangled and unmaintainable.

## LANGUAGE-INDEPENDED PRINCIPLES

Some principles of “good” programming can be applied independently of programming language. Some of them are as follows.

### Separation of concerns and single responsibility

*Separation of concerns* (SoP) is one of the fundamental principles of engineering and science, and, in particular, software engineering (Dijkstra, 1974; Reade, 1989). The main idea of this principle is in splitting a computer code into distinct parts that overlap in functionality as little as possible. Technically, in a variety of programming paradigms, this is achieved in different ways. But it does not work automatically. Developer must make this split in his mind before implementing program code.

The *single responsibility principle* (SRP) was introduced by Martin (2002) as a part of his Principles of Object Oriented Design. Martin (2002) based it on the principle of cohesion. Although the principle is formulated in relation to the object-oriented programming paradigm, it can be successfully applied to procedural paradigm, for example for the Fortran. The main idea of this principle is that every procedure (function) $P_1$ should have a single responsibility (should be responsible for only one, strongly defined, task $T_1$). And all its services should be narrowly aligned with that responsibility (every other procedures (functions) $P_2$ .. $P_n$, which use it, should use it only for solving task $T_1$). Both, SoP and SRP are very close to each other and always work together. For

example, single function should never make some calculations and perform writing to files.

**Naming Conventions**

Naming convention is a set of rules for choosing the character sequence to be used for identifiers (variables, functions, etc.) in a source code (and documentation). Importance of naming conventions in the coding is very often underestimated. Correct and uniform naming style makes code much more understandable. IDE helps much in operating with long-named objects. One of the mnemonics for naming look like: “if you want to add comment to the function, but instead think about how to rename it”. So naming should be self-documenting. It is possible if separation of concerns and single responsibility approaches are used.

**Static memory and global variables**

Using static memory is totally obsolete. Modern compilers have all possibilities to work with dynamic memory. Static memory should never be used. Global variables make program extremely tangled. Furthermore, they could prevent compilers of using some optimization techniques.

**Goto statement**

Goto statement is also obsolet. In the most cases it could be replaced with other structures. Using this construction leads to very tangled code.

## PARALLEL COMPUTING

Nowadays parallel computing is widely spread, almost every scientific laboratory has a cluster consisting of many computers with multiple cores. But efficient utilization of these resources requiers deep understanding of how it works. Modern compilers still cannot do parallelization an an optimal way. Different architectures of parallel computers lead to different parallel programming technologies. For example, OpenMP (Open Multi Processing) is suitable for shared memory systems, MPI (Message Passing Interface) is better for distributed memory architectures. But modern supercomputers usually unite both shared and distributed memory approach. This means that better performance could be achieved with using both of these technologies together.

Beside ”classical” parallel technologies new ones appeared recently. One of them is based on GPGPU (General-Purpose Graphics Processing Units). Due to computer graphics needs, GPUs could perform huge amount of simple floating point operations simultaneously, making them very fast. Using these devices for numerical methods could significantly increase calculation speed in many cases. But effective programming of such devices is a very challenge task. Only personal experience and creativity can lead to high-efficient codes.

## FUTURE WAYS OF IMPROVING SPEED

Very recently a computer with a theoretical chip, incorporating fuzzy logic was introduced (Benett, 2011). A fuzzy (or sloppy) chip can perform less accurate but significantly faster calculations. It seems to be very perspective for numerical analysis.

## AN EXAMPLE

Current project in which author is involved deals in particular with refactoring of 3-D forward and inversion codes (Zhang et al, 2012) based on integral equation approach. Refactoring and parallelization of the inverse and forward code resulted in 120 times speed up compared with initial performance of the code.

## CONCLUSION

The effective coding often could deliver more efficiency to the EM programs than the implementation of the advanced numerical algorithms. Effective coding also allows for: a) easy maintaining of the code; b) introducing new features; c) avoiding the frustration during writing and testing the codes. Unfortunately small attention is still paid in EM community to effective coding, especially at educational level. The synergy of scientists and professional programmers is needed for achieving significant progress in the field of 3-D electromagnetics (Figure 1).

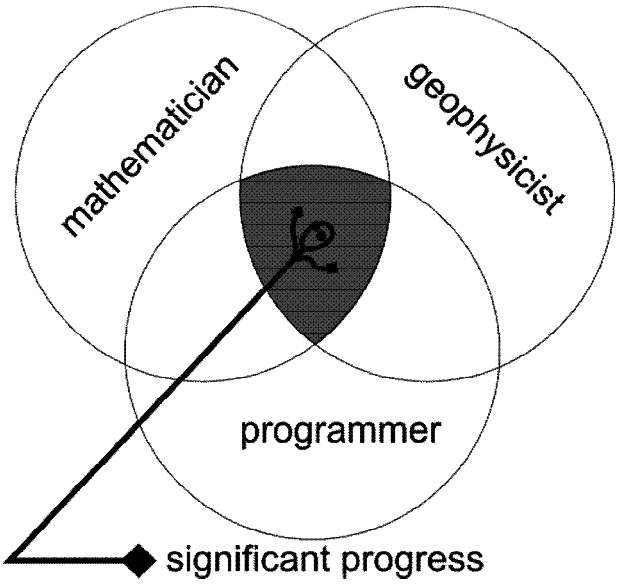


**Figure 1.** The way for achieving significant progress in the field of 3-D electromagnetics.

**ACKNOWLEDGMENTS**

This study was conducted during post-doctoral fellowship of author in the Earthquake Research Institute, the University of Tokyo. Author especially thanks Hisashi Utada who kindly offered this position to him. This work is partially supported by JSPS KAKENHI (#22000003). During this study the computer system of the Earthquake Informational Center of the Earthquake Research Institute, the University of Tokyo, was used. Author also thanks Alexey Kuvshinov for motivating him to write this abstract.

**REFERENCES**

Dijkstra, E. W., 1982, On the role of scientific thought, Computing: A Personal Perspective, p. 60–66.

Fowler M., 1999, Refactoring. Improving the Design of Existing Code. Addison-Wesley.

Gamma E., Helm R., Jhonson R., Vlissides J., 1995, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley.

Goldberg A., Kay A., 1976, Smalltalk-72 Instruction Manual. Palo Alto, California: Xerox Palo Alto Research Center.

Hardesty, L.., 2011, The surprising usefulness of sloppy arithmetic, MIT News Office.

Holmevik, J.R., 1994, Compiling SIMULA: a historical study of technological genesis, Annals of the History of Computing, IEEE , vol.16, no.4, pp.25-37

Joseph, M. J., Frank, M. G., 1988, Juran's Quality Control Handbook, McGraw-Hill

Martin, C. M., 2002, Agile Software Development, Principles, Patterns, and Practices. Prentice Hall.

Reade, C., 1989, Elements of Functional Programming, Boston, MA, USA: Addison-Wesley Longman Publishing Co., pp. 600.

Worth, D. J., 2008, State of the art in object oriented programming with fortran, Technical Report RAL-TR-2008-002.

Zhang, L., Koyama, T., Utada, H., Yu, P. and Wang, J., 2012, A regularized three-dimensional magnetotelluric inversion with a minimum gradient support constraint. Geophysical Journal International, 189: 296–316.